# C51RF-3-PK

# 完全实验手册

**成都无线龙通讯科技有限公司**

**2008** 年 **6** 月

# 目 录

# 1、CC2430 处理器基础实验

## 1.1、CC2430 基础实验一自动闪烁

### 1.1.1、实验介绍

本次实验的目的是让用户学会使用 CC2430 的 I/0 来控制外设，本例以 LED 灯为外设，用 CC2430 控制简单外设时，应将 I/O 设置为输出。实验现象是绿色 LED 闪烁。

### 1.1.2、 实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR，没有设置而是取默认认值的寄存器有：P1SEL,P1INP。

P1 （P1 口寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | P1[7:0] | 0x00 | 可读/写 | P1 端口普通功能寄存器，可位寻址 |

P1DIR （P1 方向寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | DIRP1_7 | 0 | 可读/写 | P1_7 方向<br>0 输入<br>1 输出 |
| 6 | DIRP1_6 | 0 | 可读/写 | P1_6 方向<br>0 输入<br>1 输出 |
| 5 | DIRP1_5 | 0 | 可读/写 | P1_5 方向<br>0 输入<br>1 输出 |
| 4 | DIRP1_4 | 0 | 可读/写 | P1_4 方向<br>0 输入<br>1 输出 |
| 3 | DIRP1_3 | 0 | 可读/写 | P1_3 方向<br>0 输入<br>1 输出 |
| 2 | DIRP1_2 | 0 | 可读/写 | P1_2 方向<br>0 输入<br>1 输出 |
| 1 | DIRP1_1 | 0 | 可读/写 | P1_1 方向 |

| | | | | |
|---|---|---|---|---|
| | | | | 0 输入<br>1 输出 |
| 0 | DIRP1_0 | 0 | 可读/写 | P1_0 方向<br>0 输入<br>1 输出 |

P1SEL （P1 功能选择寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | SELP1_7 | 0 | 可读/写 | P1_7 功能<br>0 普通 I/O<br>1 外设功能 |
| 6 | SELP1_6 | 0 | 可读/写 | P1_6 功能<br>0 普通 I/O<br>1 外设功能 |
| 5 | SELP1_5 | 0 | 可读/写 | P1_5 功能<br>0 普通 I/O<br>1 外设功能 |
| 4 | SELP1_4 | 0 | 可读/写 | P1_4 功能<br>0 普通 I/O<br>1 外设功能 |
| 3 | SELP1_3 | 0 | 可读/写 | P1_3 功能<br>0 普通 I/O<br>1 外设功能 |
| 2 | SELP1_2 | 0 | 可读/写 | P1_2 功能<br>0 普通 I/O<br>1 外设功能 |
| 1 | SELP1_1 | 0 | 可读/写 | P1_1 功能<br>0 普通 I/O<br>1 外设功能 |
| 0 | SELP1_0 | 0 | 可读/写 | P1_0 功能<br>0 普通 I/O<br>1 外设功能 |

## 1.1.3、实验相关函数

写在程序中子函数及功能列写如下：
void Delay(uint n)；
函数原型是
void Delay(uint n)

```
{
	uint tt;
	for(tt = 0;tt<n;tt++);
```

```
	for(tt = 0;tt<n;tt++);
	for(tt = 0;tt<n;tt++);
	for(tt = 0;tt<n;tt++);
	for(tt = 0;tt<n;tt++);
}
```

函数功能是软件延时，执行 5 次 0 到 n 的空循环来实现软件延时。延时时间约为 5*n/32 μs。

void Initial(void)；
函数原型是：
void Initial(void)

```
{
	P1DIR |= 0x03; //P10、P11 定义为输出

	RLED = 1;
	YLED = 1; //LED
}
```

函数功能是把连接 LED 的两个 I/0 设置为输出，同时将它们设为高电平（此时 LED 灭）。

# 1.2、CC2430 基础实验二　按键控制开关

## 1.2.1、实验介绍

让用户掌握 CC2430 的按键应用这一常用人机交互方法，本次实用两个分别控制两个 LED 灯。

## 1.2.2、实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR，P1SEL, **P1INP**。前面三个寄存器在**实验**一已经有详述，这里不再重复介绍。

**P1** 参见**实验**一说明文档
**P1DIR** 参见**实验**一说明文档
**P1SEL** 参见**实验**一说明文档

**P1INP** （P1 输入模式寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
| --- | --- | --- | --- | --- |
| 7 | PDUP2 | 0 | 可读/写 | P2 口上/下拉选择<br>0 上拉<br>1 下拉 |
| 6 | PDUP1 | 0 | 可读/写 | P1 口上/下拉选择 |

| | | | | |
|---|---|---|---|---|
| | | | | 0 上拉<br>1 下拉 |
| 5 | PDUP0 | 0 | 可读/写 | P0 口上/下拉选择<br>0 上拉<br>1 下拉 |
| 4 | MDP2_4 | 0 | 可读/写 | P2_4 输入模式<br>0 上拉<br>1 下拉 |
| 3 | MDP2_3 | 0 | 可读/写 | P2_3 输入模式<br>0 上拉<br>1 下拉 |
| 2 | MDP2_2 | 0 | 可读/写 | P2_2 输入模式<br>0 上拉<br>1 下拉 |
| 1 | MDP2_1 | 0 | 可读/写 | P2_1 输入模式<br>0 上拉<br>1 下拉 |
| 0 | MDP2_0 | 0 | 可读/写 | P2_0 输入模式<br>0 上拉<br>1 下拉 |

## 1.2.3、实验相关函数

写在程序中子函数及功能列写如下：

void Delay(uint n)；参见 CC2430 基础实验一。

void Initial(void)；参见 CC2430 基础实验一。

void InitKey(void);
函数原型：

```
void InitKey(void)
{
   P1SEL &= ~0X0C;    //P12,P13 输入
   P1DIR &= ~0X0C;    //按键在 P12 P13
   P1INP |= 0x0c;         //三态
}
```

函数功能是将 I/O P1_2,P1_3 设为输入（且为三态）以读取按键的状态。

unsigned char KeyScan(void);
函数原型：

```
uchar KeyScan(void)
{
  if(K1 == 0)          //低电平有效
  {
    Delay(100);        //检测到按键
    if(K1 == 0)        //前面定义了 #define K1 P1_2
    {
      while(!K1);      //直到松开按键
      return(1);
    }

  }
  if(K2 == 0)
  {
    Delay(100);
    if(K2 == 0)
    {
      while(!K2);
      return(2);
    }
  }
  return(0);
}
```

函数功能是检测按键是否按下，若有键按下，则返回相应的值，如 P1_2 对应的按键按下则返回 1，P1_3 对应的按键按下返回 2。

# 1.3、CC2430 基础实验三　按键控制闪烁

## 1.3.1、实验介绍

本实验的控制比实验二的控制稍显复杂，这个实验中使用按键控制 LED 或闪烁，或熄灭。

## 1.3.2、 实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR，P1SEL,P1INP。前面三个寄存器在实验一已经有详述，这里不再重复介绍。

**P1** 参见实验一说明文档
**P1DIR** 参见实验一说明文档
**P1SEL** 参见实验一说明文档

**P1INP** 参见实验二说明文档

## 1.3.3、 实验相关函数

写在程序中子函数及功能列写如下：

**void Delay(uint n)**；参见 CC2430 基础实验一。

**void Initial(void)**；参见 CC2430 基础实验一。

**void InitKey(void)**;参见 CC2430 基础实验二。

unsigned char KeyScan(void); 参见 CC2430 基础实验二。

# 1.4、CC2430 基础实验四 T1 的使用

## 1.4.1、实验介绍

用定时器 1 来改变小灯的状态，T1 每溢出两次，两个小灯闪烁一次，并且在停止闪烁后成闪烁前相反的状态。

## 1.4.2、实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR，P1SEL, T1CTL 前面三个寄存器在实验一已经有详述，这里不再重复介绍。

**P1** 见实验一说明文档
**P1DIR** 见实验一说明文档
**P1SEL** 见实验一说明文档

T1CTL（T1 控制&状态寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | CH2IF | 0 | 可读/写 0 | 定时器 1 通道 2 中断标志位 |
| 6 | CH1IF | 0 | 可读/写 0 | 定时器 1 通道 1 中断标志位 |
| 5 | CH0IF | 0 | 可读/写 0 | 定时器 1 通道 0 中断标志位 |
| 4 | OVFIF | 0 | 可读/写 0 | 定时器溢出中断标志， |

| | | | | 在在计数器达到计数终值的时候置位 |
|---|---|---|---|---|
| 3:2 | DIV[1:0] | 00 | 可读/写 | 定时器 1 计数时钟分步选择<br>00 不分频<br>01 8 分频<br>10 32 分频<br>11 128 分频 |
| 1:0 | MODE[1:0] | 00 | 可读/写 | 定时器 1 模式选择<br>00 暂停<br>01 自动重装 0x0000-0xffff<br>10 比较计数 0x0000-T1CC0<br>11 PWM 方式 0x0000-T1CC0-0X0000 |

## 1.4.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(uint n)；参见 CC2430 基础实验一。

void Initial(void)；
函数原型：
void Initial(void)

```
{
    //初始化 P1
    P1DIR = 0x03;   //P10 P11 为输出
    RLED = 1;
    YLED = 1;       //灭 LED

    //用 T1 来做实验
    T1CTL = 0x3d;   // 128 分频;自动重装模式(0x0000->0xffff);
}
```

函数功能是将 P10,P11 设为输出，并将定时器 1 设为自动重装模式，计数时钟为 0.25M。

# 1.5、CC2430 基础实验五 T2 的使用

## 1.5.1、实验介绍

用定时器 2 来改变小灯的状态，T2 每发生一次中断小灯改变状态一次。

## 1.5.2、 实验相关寄存器

实验中操作了的寄存器有 P1,P1SEL,P1DIR,T2CNF, T2PEROF2，T2CAPLPL，T2CAPLPH，IEN0 等寄存器。

**P1** 参见 CC2430 实验一
**P1SEL** 参见 CC2430 实验一
**P1DIR** 参见 CC2430 实验一

T2CNF（T2 配置寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
| --- | --- | --- | --- | --- |
| 7 | CMPIF | 0 | 可读/写 0 | 定时器 2 比较中断标志，当比较中断发生硬件置 1，只能由软件清除，写 1 无效 |
| 6 | PERIF | 0 | 可读/写 0 | 定时器 2 溢出中断标志，当一个周期事件发生时硬件置 1，只能由软件清除，写 1 无效 |
| 5 | OFCMPIF | 0 | 可读/写 0 | T2 溢出比较中断标志，当一个溢出比较事件发生时硬件置 1，只能由软件清除，写 1 无效 |
| 4 | — | 0 | 读 0 | 没用，读出为 0 |
| 3 | CMSEL | 0 | 可读/写 | T2 比较目标设置<br>0 取 T2 计数值高 8 位[15：8]<br>1 取 T2 计数值低 8 位[7：0] |
| 2 | — | 0 | R/W | 保留，设置为 0 |
| 1 | SYNC | 1 | R/W | 同步使能<br>0 T2 立即起、停<br>1 T2 起、停和 32.768kHz 时钟及计数新值同步 |
| 0 | RUN | 0 | R/W | 启动 T2，通过读出该位可以知道 T2 的状态 |

| | | | | |
|---|---|---|---|---|
| | | | | 0 停止 T2（IDLE）<br>1 启动 T2（RUN） |

T2PEROF2 （T2 溢出计数器 2 寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | CMPIM | 0 | R/W | 比较中断掩码<br>0 关中断<br>1 开中断 |
| 6 | PERIM | 0 | R/W | 溢出中断掩码<br>0 关中断<br>1 开中断 |
| 5 | OFCMPIM | 0 | R/W | 溢出计数比较中断掩码<br>0 关中断<br>1 开中断 |
| 4 | - | 0 | R0 | 没有，读出为 0 |
| 3:0 | PEROF2[3:0] | 0000 | R/W | 溢出计数捕获/溢出计数比较值，写值到这 4 位设置溢出计数比较值的 19－16 位，读这 4 位的值得到最后一次发生捕获事件时溢出计数值的 19－16 位。 |

T2CAPHPH（T2 周期寄存器高字节）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | CMPIM | 0xff | R/W | 捕获值/时间周期值高字节，写该寄存器设定 T2 周期时间[15-8]位，读该寄存器得到后一次发生捕获事件时溢出计数值的[15－8]位。 |

T2CAPLPL (T2 周期寄存器低字节)

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | CAPHPH | 0xff | R/W | 捕获值/时间周期值低字节，写该寄存器设定 T2 周期时间[7－0]位，读该寄存器得到后一次发生捕获事件时溢出计数值的[7－0]位。 |

IEN0（中断使能控制寄存 0）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | EAL | 0 | R/W | 总中断使能 |

| | | | | |
|---|---|---|---|---|
| | | | | 0 禁止所有中断<br>1 允许中断 |
| 6 | - | 0 | R0 | 未用，读出为 0 |
| 5 | STIE | 0 | R/W | 睡眠定时器中断使能<br>0 关中断<br>1 开中断 |
| 4 | ENCIE | 0 | R/W | AES 加解密中断使能<br>0 关中断<br>1 开中断 |
| 3 | URX1IE | 0 | R/W | 串口 1 接收中断使能<br>0 关中断<br>1 开中断 |
| 2 | URX0IE | 0 | R/W | 串口 0 接收中断使能<br>0 关中断<br>1 开中断 |
| 1 | ADCIE | 0 | R/W | ADC 中断使能<br>0 关中断<br>1 开中断 |
| 0 | RFERRIE | 0 | R/W | 射频 TX/RX FIFO 中断<br>0 关中断<br>1 开中断 |

## 1.5.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(uint n)；参见 CC2430 基础**实验**一。

void Initial(void)；
函数原型：

```
void Initial(void)
{
        LED_ENALBLE(); //启用 LED
          //用 T2 来做实验
        SET_TIMER2_OF_INT();                    //开溢出中断
        SET_TIMER2_CAP_COUNTER(0X00ff);
}
```

函数功能是启用 LED，使用 LED 可控，开 T2 比较中断

### 1.5.3、 重要的宏定义

开启溢出中断

```
#define SET_TIMER2_CAP_INT()           \
   do{                                 \
      EA = 1;                          \
      T2IE = 1;                        \
      T2PEROF2 |= 0x40;                \
}while(0)
```

设定溢出周期

```
#define SET_TIMER2_OF_COUNTER(val) SET_WORD(T2CAPLPL,T2CAPHPH,val)
```

功能：将无符号整形数 val 的高 8 位写入 T2CAPLPL，低 8 位写入 T2CAPHPH。

启动 T2

```
#define TIMER2_RUN()    T2CNF|=0X01
```

停止 T2

```
#define TIMER2_STOP() do{T2CNF&=0XFE;}while(0)
```

## 1.6、CC2430 基础实验六　T3 的使用

### 1.6.1、实验介绍

用定时器 3 来改变小灯的状态，T3 每发生 200 次中断小灯改变状态一次。

### 1.6.2、 实验相关寄存器

实验中操作了的寄存器有 P1,P1SEL,P1DIR,T3CTL,T3CCTL0,T3CC0,T3CCTL1,T3CC1,等寄存器。

**P1** 参见 CC2430 实验一
**P1SEL** 参见 CC2430 实验一
**P1DIR** 参见 CC2430 实验一

T3CTL（T3 控制寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:5 | DIV[2:0] | 000 | R/W | 定时器时钟再分频数（对 CLKCON.TICKSPD 分频后再次分频） |

| | | | | 000 不再分频<br>001 2 分频<br>010 4 分频<br>011 8 分频<br>100 16 分频<br>101 32 分频<br>110 64 分频<br>111 128 分频 |
|---|---|---|---|---|
| 4 | START | 0 | R/W | T3 起停位<br>0 暂停计数<br>1 正常运行 |
| 3 | OVFIM | 1 | R/W0 | 溢出中断掩码<br>0 关溢出中断<br>1 开溢出中断 |
| 2 | CLR | 0 | R0/W1 | 清计数值，写 1 使 T3CNT=0x00 |
| 1:0 | MODE[1:0] | 00 | R/W | T3 模式选择<br>00 自动重装<br>01 DOWN （从 T3CC0 到 0x00 计数一次）<br>10 模计数（反复从 0x00 到 T3CC0 计数）<br>11 UP/DOWN（反复从 0x00 到 T3CC0 再到 0x00） |

T3CCTL0 （T3 通道 0 捕获/比较控制寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | 一 | 0 | R0 | 没用 |
| 6 | IM | 1 | R/W | 通道 0 中断掩码<br>0 关中断<br>1 开中断 |
| 5:3 | CMP[7:0] | 000 | R/W | 通道 0 比较输出模式选择，指定计数值过 T3CC0 时的发生事件<br>000 输出置 1（发生比较时）<br>001 输出清 0（发生比较时）<br>010 输出翻转<br>011 输出置 1（发生上比较时）输出清 0（计数值为 0 或 UP/DOWN 模式下 |

| | | | | |
|---|---|---|---|---|
| | | | | 发生下比较）<br>100 输出清 0（发生上比较时）输出置 1（计数值为 0 或 UP/DOWN 模式下发生下比较）<br>101 输出置 1（发生比较时）输出清 0（计数值为 0xff 时）<br>110 输出清 0（发生比较时）输出置 1（计数值为 0x00 时）<br>111 没用 |
| 2 | MODE- | 0 | R/W | T3 通道 0 模式选择<br>0 捕获<br>1 比较 |
| 1:0 | CAP | 00 | R/W | T3 通道 0 捕获模式选择<br>00 没有捕获<br>01 上升沿捕获<br>10 下降沿捕获<br>11 边沿捕获 |

T3CC0 （T3 通道 0 捕获/比较值寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | VAL[7:0] | 0x00 | R/W | T3 通道 0 比较/捕获值 |

T3CCTL1 （T3 通道 1 捕获/比较控制寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | — | 0 | R0 | 没用 |
| 6 | IM | 1 | R/W | 通道 1 中断掩码<br>0 关中断<br>1 开中断 |
| 5:3 | CMP[7:0] | 0 | R/W | 通道 1 比较输出模式选择，指定计数值过 T3CC0 时的发生事件<br>000 输出置 1（发生比较时）<br>001 输出清 0（发生比较时）<br>010 输出翻转<br>011 输出置 1（发生上比较时）输出清 0（计数值为 0 或 UP/DOWN 模式下发生下比较）<br>100 输出清 0（发生上比 |

| | | | | |
|---|---|---|---|---|
| | | | | 较时）输出置 1（计数值为 0 或 UP/DOWN 模式下发生下比较）<br>101 输出置 1（发生比较时）输出清 0（计数值为 0xff 时）<br>110 输出清 0（发生比较时）输出置 1（计数值为 0x00 时）<br>111 没用 |
| 2 | MODE- | 0 | R/W | T3 通道 1 模式选择<br>0 捕获<br>1 比较 |
| 1:0 | CAP | 0000 | R/W | T3 通道 1 捕获模式选择<br>00 没有捕获<br>01 上升沿捕获<br>10 下降沿捕获<br>11 边沿捕获 |

T3CC1 （T3 通道 1 捕获/比较值寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | VAL[7:0] | 0x00 | R/W | T3 通道 1 比较/捕获值 |

## 1.6.3、 实验相关函数

写在程序中的子函数及功能列写如下：

void Init_T3_AND_LED(void)；
函数原型：

```
void Init_T3_AND_LED(void)
{
    P1DIR = 0X03;
    RLED = 1;
    YLED = 1;

    TIMER34_INIT(3);                        //初始化 T3
    TIMER34_ENABLE_OVERFLOW_INT(3,1);   //开 T3 中断

    //时钟 32 分频 101
    TIMER3_SET_CLOCK_DIVIDE(16);
    TIMER3_SET_MODE(T3_MODE_FREE);                   //自动重装 00－>0xff
```

```
    TIMER3_START(1);                         //启动
};
```

函数功能：将 I/0 P10,P11 设置为输出去控制 LED，将 T3 设置为自动重装模式，定时器时钟 16 分频，并启动 T3。

void T3_ISR(void)；

函数原型：

```
#pragma vector = T3_VECTOR
 __interrupt void T3_ISR(void)
 {
      //IRCON = 0x00;                  //清中断标志,硬件自动完成
          if(counter<200)counter++;    //10 次中断 LED 闪烁一轮
          else
          {
            counter = 0;                     //计数清零
          RLED = !RLED;                            //改变小灯的状态
          }
 }
```

函数功能：这是一个中断服务程序，每 200 次中断改变一次红色 LED 的状态。

## 1.6.4、 重要的宏定义

开启溢出中断

#define TIMER34_ENABLE_OVERFLOW_INT(timer,val) \

```
  do{T##timer##CTL =   (val) ? T##timer##CTL | 0x08 : T##timer##CTL & ~0x08; \
    EA = 1;                                                                   \
    T3IE = 1;                                                                 \
    }while(0)
```

功能：打开 T3 的溢出中断。

复位 T3 相关寄存器

```
#define TIMER34_INIT(timer)     \
   do {                                \
      T##timer##CTL     = 0x06; \
      T##timer##CCTL0 = 0x00; \
      T##timer##CC0     = 0x00; \
      T##timer##CCTL1 = 0x00; \
      T##timer##CC1     = 0x00; \
   } while (0)
```

功能：将 T3 相关的寄存器复位到 0

控制 T3 起停

```
#define TIMER3_START(val)                                    \
    (T3CTL = (val) ? T3CTL | 0X10 : T3CTL&~0X10)
```

功能：val 为 1，T3 正常运行，val 为 0，T3 停止计数

设置 T3 工作方式

```
#define TIMER3_SET_MODE(val)                                   \
   do{                                                         \
      T3CTL &= ~0X03;                                          \
      (val==1)?(T3CTL|=0X01):   /*DOWN                 */ \
      (val==2)?(T3CTL|=0X02):   /*Modulo               */ \
      (val==3)?(T3CTL|=0X03):   /*UP / DOWN            */ \
      (T3CTL|=0X00);            /*free runing */          \
   }while(0)

#define T3_MODE_FREE       0X00
#define T3_MODE_DOWN       0X01
#define T3_MODE_MODULO     0X02
#define T3_MODE_UP_DOWN 0X03
```

功能：根据 val 的值将 T3 设置为不同模式，一共 4 种模式。

# 1.7、CC2430 基础实验七　T4 的使用

## 1.7.1、实验介绍

用定时器 4 来改变小灯的状态，T4 每发生 200 次中断小灯改变状态一次。

## 1.7.2、 实验相关寄存器

实验中操作了的寄存器有 P1,P1SEL,P1DIR,T4CTL,T4CCTL0,T4CC0,T4CCTL1,T4CC1,等寄存器。

**P1**　　参见 CC2430 实验一
**P1SEL**　参见 CC2430 实验一
**P1DIR**　参见 CC2430 实验一

T4CTL（T4 控制寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:5 | DIV[2:0] | 000 | R/W | 定时器时钟再分频数（对 CLKCON.TICKSPD 分频后再次分频）<br>000 不再分频<br>001 2 分频<br>010 4 分频 |

| | | | | |
|---|---|---|---|---|
| | | | | 011 8 分频<br>100 16 分频<br>101 32 分频<br>110 64 分频<br>111 128 分频 |
| 4 | START | 0 | R/W | T4 起停位<br>0 暂停计数<br>1 正常运行 |
| 3 | OVFIM | 1 | R/W0 | 溢出中断掩码<br>0 关溢出中断<br>1 开溢出中断 |
| 2 | CLR | 0 | R0/W1 | 清计数值，写 1 使 T4CNT=0x00 |
| 1:0 | MODE[1:0] | 00 | R/W | T4 模式选择<br>00 自动重装<br>01 DOWN （从 T4CC0 到 0x00 计数一次）<br>10 模计数（反复从 0x00 到 T4CC0 计数）<br>11 UP/DOWN（反复从 0x00 到 T4CC0 再到 0x00） |

T4CCTL0 （T4 通道 0 捕获/比较控制寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | — | 0 | R0 | 没用 |
| 6 | IM | 1 | R/W | 通道 0 中断掩码<br>0 关中断<br>1 开中断 |
| 5:3 | CMP[7:0] | 000 | R/W | 通道 0 比较输出模式选择,指定计数值过 T4CC0 时的发生事件<br>000 输出置 1（发生比较时）<br>001 输出清 0（发生比较时）<br>010 输出翻转<br>011 输出置 1（发生上比较时）输出清 0（计数值为 0 或 UP/DOWN 模式下发生下比较）<br>100 输出清 0（发生上比较时）输出置 1（计数值 |

| | | | | |
|---|---|---|---|---|
| | | | | 为0或UP/DOWN模式下发生下比较）<br>101 输出置1（发生比较时）输出清0（计数值为0xff时）<br>110 输出清0（发生比较时）输出置1（计数值为0x00时）<br>111 没用 |
| 2 | MODE- | 0 | R/W | T4通道0模式选择<br>0 捕获<br>1 比较 |
| 1:0 | CAP | 00 | R/W | T4通道0捕获模式选择<br>00 没有捕获<br>01 上升沿捕获<br>10 下降沿捕获<br>11 边沿捕获 |

T4CC0 （T4通道0捕获/比较值寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | VAL[7:0] | 0x00 | R/W | T4通道0比较/捕获值 |

T4CCTL1 （T4通道1捕获/比较控制寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | 一 | 0 | R0 | 没用 |
| 6 | IM | 1 | R/W | 通道1中断掩码<br>0 关中断<br>1 开中断 |
| 5:3 | CMP[7:0] | 0 | R/W | 通道 1 比较输出模式选择，指定计数值过T4CC0时的发生事件<br>000 输出置1（发生比较时）<br>001 输出清0（发生比较时）<br>010 输出翻转<br>011 输出置1（发生上比较时）输出清0（计数值为0或UP/DOWN模式下发生下比较）<br>100 输出清 0（发生上比较时）输出置1（计数值为0或UP/DOWN模式下发生下比较） |

| | | | | |
|---|---|---|---|---|
| | | | | 101 输出置 1（发生比较时）输出清 0（计数值为 0xff 时）<br>110 输出清 0（发生比较时）输出置 1（计数值为 0x00 时）<br>111 没用 |
| 2 | MODE- | 0 | R/W | T4 通道 1 模式选择<br>0 捕获<br>1 比较 |
| 1:0 | CAP | 0000 | R/W | T4 通道 1 捕获模式选择<br>00 没有捕获<br>01 上升沿捕获<br>10 下降沿捕获<br>11 边沿捕获 |

T4CC1 （T4 通道 1 捕获/比较值寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | VAL[7:0] | 0x00 | R/W | T4 通道 1 比较/捕获值 |

## 1.7.3、 实验相关函数

写在程序中的子函数及功能列写如下：

void Init_T4_AND_LED(void)；
函数原型：

```
void Init_T4_AND_LED(void)
{
    P1DIR = 0X03;
    led1 = 1;
    led2 = 1;

    TIMER34_INIT(4);                          //初始化 T4
    TIMER34_ENABLE_OVERFLOW_INT(4,1);   //开 T4 中断

    TIMER34_SET_CLOCK_DIVIDE(4,128);
    TIMER34_SET_MODE(4,0);                        //自动重装 00－>0xff

    TIMER34_START(4,1);                           //启动
};
```

函数功能：将 I/0 P10,P11 设置为输出去控制 LED，将 T4 设置为自动重装模式，定时器时钟

16 分频，并启动 T4。

void T4_ISR(void)；

函数原型：

```
#pragma vector = T4_VECTOR
 __interrupt void T4_ISR(void)
 {
      //IRCON = 0x00;                  //清中断标志,硬件自动完成
          if(counter<200)counter++;    //10 次中断 LED 闪烁一轮
         else
         {
           counter = 0;                  //计数清零
          RLED = !RLED;                       //改变小灯的状态
         }
 }
```

函数功能：这是一个中断服务程序，每 200 次中断改变一次红色 LED 的状态。

## 1.7.4、 重要的宏定义

开启溢出中断

```
#define TIMER34_ENABLE_OVERFLOW_INT(timer,val) \
  do{T##timer##CTL =   (val) ? T##timer##CTL | 0x08 : T##timer##CTL & ~0x08; \
    EA = 1;                                                                     \
    T4IE = 1;                                                                  \
    }while(0)
```

功能：打开 T4 的溢出中断。

复位 T4 相关寄存器

```
#define TIMER34_INIT(timer)     \
   do {                           \
       T##timer##CTL     = 0x06; \
       T##timer##CCTL0 = 0x00; \
       T##timer##CC0     = 0x00; \
       T##timer##CCTL1 = 0x00; \
       T##timer##CC1     = 0x00; \
   } while (0)
```

功能：将 T4 相关的寄存器复位到 0

控制 T4 起停

```
#define TIMER#define TIMER34_START(timer,val)                          \
    (T##timer##CTL = (val) ? T##timer##CTL | 0X10 : T##timer##CTL&~0X10)
```

功能：timer 为定时器序号，只能取 3 或 4。val 为 1，定时器正常运行，val 为 0，定时器停止计数

设置 T4 工作方式

```
#define TIMER3_SET_MODE(val)                          \
   do{                                                \
      T4CTL &= ~0X03;                                 \
      (val==1)?(T4CTL|=0X01):   /*DOWN          */ \
      (val==2)?(T4CTL|=0X02):   /*Modulo        */ \
      (val==3)?(T4CTL|=0X03):   /*UP / DOWN     */ \
      (T4CTL|=0X00);            /*free runing */    \
   }while(0)

#define T4_MODE_FREE       0X00
#define T4_MODE_DOWN       0X01
#define T4_MODE_MODULO     0X02
#define T4_MODE_UP_DOWN 0X03
```

功能：根据 val 的值将 T4 设置为不同模式，一共 4 种模式。

# 1.8、CC2430 基础实验八　定时器中断

## 1.8.1、实验介绍

用定时器 4 来改变小灯的状态，T4 每 2000 次中断小灯闪烁一轮，闪烁的时间长度为 1000 次中断所耗时间。

## 1.8.2、 实验相关寄存器

实验中操作了的寄存器有 P1,P1SEL,P1DIR,T4CTL,T4CCTL0,T4CC0,T4CCTL1,T4CC1，IEN0,IEN1 等寄存器。

**P1** 参见 CC2430 实验一
**P1SEL** 参见 CC2430 实验一
**P1DIR** 参见 CC2430 实验一

**T4CTL** 参见 CC2430 实验七
**T4CCTL0** 参见 CC2430 实验七
**T4CC0** 参见 CC2430 实验七
**T4CCTL1** 参见 CC2430 实验七
**T4CC1** 参见 CC2430 实验七

## 1.8.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Init_T4_AND_LED(void)；
函数原型：

```
void Init_T4_AND_LED(void)
{
    P1DIR = 0X03;
    led1 = 1;
    led2 = 1;

    TIMER34_INIT(4);                        //初始化 T4
    TIMER34_ENABLE_OVERFLOW_INT(4,1);   //开 T4 中断

    TIMER34_SET_CLOCK_DIVIDE(4,128);
    TIMER34_SET_MODE(4,0);                      //自动重装 00－>0xff

    TIMER34_START(4,1);                         //启动
};
```

函数功能：将 I/0 P10,P11 设置为输出去控制 LED，将 T4 设置为自动重装模式，定时器时钟 16 分频，并启动 T4。

void T4_ISR(void)；
函数原型：

```
#pragma vector = T4_VECTOR
 __interrupt void T4_ISR(void)
 {
     IRCON = 0x00;              //可不清中断标志,硬件自动完成
          if(counter<1000)counter++;  //10 次中断 LED 闪烁一轮
         else
         {
          counter = 0;                      //计数清零
          GlintFlag = !GlintFlag;               //GlintFalg = 1，LED 闪烁
         }
 }
```

函数功能：这是一个中断服务程序，每 1000 次中断改变一次红色 LED 的状态。

## 1.8.3、 重要的宏定义

开启溢出中断
#define TIMER34_ENABLE_OVERFLOW_INT(timer,val) \

```
  do{T##timer##CTL =   (val) ? T##timer##CTL | 0x08 : T##timer##CTL & ~0x08; \
```

```
    EA = 1;                                                              \
    T4IE = 1;                                                            \
    }while(0)
```

功能：打开 T4 的溢出中断。

复位 T4 相关寄存器

```
#define TIMER34_INIT(timer)      \
   do {                           \
      T##timer##CTL      = 0x06; \
      T##timer##CCTL0 = 0x00; \
      T##timer##CC0      = 0x00; \
      T##timer##CCTL1 = 0x00; \
      T##timer##CC1      = 0x00; \
   } while (0)
```

功能：将 T4 相关的寄存器复位到 0

控制 T4 起停

```
#define TIMER#define TIMER34_START(timer,val)                              \
    (T##timer##CTL = (val) ? T##timer##CTL | 0X10 : T##timer##CTL&~0X10)
```

功能：timer 为定时器序号，只能取 3 或 4。val 为 1，定时器正常运行，val 为 0，定时器停止计数

设置 T4 工作方式

```
#define TIMER3_SET_MODE(val)                         \
  do{                                                \
    T4CTL &= ~0X03;                                  \
    (val==1)?(T4CTL|=0X01):   /*DOWN           */ \
    (val==2)?(T4CTL|=0X02):   /*Modulo         */ \
    (val==3)?(T4CTL|=0X03):   /*UP / DOWN      */ \
    (T4CTL|=0X00);              /*free runing */     \
  }while(0)

#define T4_MODE_FREE       0X00
#define T4_MODE_DOWN       0X01
#define T4_MODE_MODULO    0X02
#define T4_MODE_UP_DOWN 0X03
```

功能：根据 val 的值将 T4 设置为不同模式，一共 4 种模式。

# 1.9、CC2430 基础实验九　外部中断

## 1.9.1、实验介绍

使用两个按键来翻转 LED 的状态，但这里两个按键不是做键盘用，而是产生中断触发信号。

## 1.9.2、 实验相关寄存器

实验中操作了的寄存器有 P1,P1SEL,P1DIR,P1INP,P1IEN,PICTL,IEN2,P1IFG 等寄存器。

**P1** 参见 CC2430 实验一
**P1SEL** 参见 CC2430 实验一
**P1DIR** 参见 CC2430 实验一
**P1INP** 参见 CC2430 实验二

P1IEN （P1 口中断掩码）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | P1_7IEN | 0 | R/W | P17 中断掩码<br>0 关中断<br>1 开中断 |
| 6 | P1_6IEN | 0 | R/W | P16 中断掩码<br>0 关中断<br>1 开中断 |
| 5 | P1_5IEN | 0 | R/W | P15 中断掩码<br>0 关中断<br>1 开中断 |
| 4 | P1_4IEN | 0 | R/W | P14 中断掩码<br>0 关中断<br>1 开中断 |
| 3 | P1_3IEN | 0 | R/W | P13 中断掩码<br>0 关中断<br>1 开中断 |
| 2 | P1_2IEN | 0 | R/W | P12 中断掩码<br>0 关中断<br>1 开中断 |
| 1 | P1_1IEN | 0 | R/W | P11 中断掩码<br>0 关中断<br>1 开中断 |
| 0 | P1_0IEN | 0 | R/W | P10 中断掩码<br>0 关中断<br>1 开中断 |

PICTL （P 口中断控制寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | — | 0 | R0 | 没用 |
| 6 | PADSC | 0 | R/W | 输出驱动能力选择<br>0 最小驱动能力<br>1 最大驱动能力 |
| 5 | P2IEN | 0 | R/W | P2（0－4）中断使能位<br>0 关中断<br>1 开中断 |
| 4 | P0IENH | 0 | R/W | P0（4－7）中断使能位<br>0 关中断<br>1 开中断 |
| 3 | P0IENL | 0 | R/W | P0（0－3）中断使能位<br>0 关中断<br>1 开中断 |
| 2 | P2ICON | 0 | R/W | P2 （0－4）中断配置<br>0 上升沿触发<br>1 下降沿触发 |
| 1 | P1ICON | 0 | R/W | P1 （0－7）中断配置<br>0 上升沿触发<br>1 下降沿触发 |
| 0 | P0ICON | 0 | R/W | P0 （0－7）中断配置<br>0 上升沿触发<br>1 下降沿触发 |

P1IFG（P1 口中断标志寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7：0 | P1IF[7:0] | 0x00 | R/W0 | P1（0-7）中断标志位，在中断条件发生，相应位自动置 1 |

IEN2 （中断使能寄存器 2）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7:6 | — | 00 | R0 | 没有，读出为 0 |
| 5 | WDTIE | 0 | R/W | 看门狗定时器中断使能<br>0 关中断<br>1 开中断 |
| 4 | P1IE | 0 | R/W | P1 中断使能使能<br>0 关中断<br>1 开中断 |
| 3 | UTX1IE | 0 | R/W | 串口 1 发送中断使能<br>0 关中断<br>1 开中断 |

| | | | | |
|---|---|---|---|---|
| 2 | UTX0IE | 0 | R/W | 串口 0 发送中断使能<br>0 关中断<br>1 开中断 |
| 1 | P2IE | 0 | R/W | P2 口中断使能<br>0 关中断<br>1 开中断 |
| 0 | RFIE | 0 | R/W | 普通射频中断使能<br>0 关中断<br>1 开中断 |

## 1.9.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Init_IO_AND_LED(void)；
函数原型：

```
void Init_IO_AND_LED(void)
{
      P1DIR = 0X03; //0 为输入（默认），1 为输出
      RLED = 1;
      GLED = 1;

      P1INP &= ~0X0c;//有上拉、下拉
      P2INP &= ~0X40; //选择上拉
      P1IEN |= 0X0c;      //P12 P13
      PICTL |= 0X02;      //下降沿
      EA = 1;
      IEN2 |= 0X10;      // P1IE = 1;

      P1IFG &= ~0x0C;      //P12 P13 中断标志清 0
}
```

函数功能：将 I/0 P10,P11 设置为输出去控制 LED，使能 P1 中断 且配置为下降沿触发。

void P1_ISR(void)；
函数原型：

```
#pragma vector = P1INT_VECTOR
 __interrupt void P1_ISR(void)
 {
            if(P1IFG>0)               //按键中断
            {
               P1IFG = 0;
               RLED = !RLED;
```

```
        }
        P1IF = 0;              //清中断标志
}
```

函数功能：在 P12，P13 触发中断的时候将红色 LED 的状态翻转。

# 1.10、CC2430 基础实验十　片内温度

## 1.10.1、实验介绍

取片内温度传感器为 AD 源，并将转换得到温度通过串口送至电脑。

## 1.10.2、实验相关寄存器

实验中操作了的寄存器有 CLKCON,SLEEP,PERCFG,U0CSR,U0GCR,U0BAUD,IEN0,U0DUB,ADCCON1,ADCCON3,ADCH,ADCL,等寄存器。

**IEN0**　参见**实验五**

CLKCON （时钟控制寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | OSC32K | 1 | R/W | 32kHz 时钟源选择<br>0 32K 晶振<br>1 32K RC 振荡 |
| 6 | OSC | 1 | R/W | 主时钟源选择<br>0 32M 晶振<br>1 16M　RC 振荡 |
| 5:3 | TICKSPD[2:0] | 001 | R/W | 定时器计数时钟分频（该时钟频不大于 OSC 决定频率）<br><br>000 32M<br>001 16M<br>010 8M<br>011 4M<br>100 2M<br>101 1M |

| | | | | |
|---|---|---|---|---|
| | | | | 110 0.5M<br>111 0.25M |
| 2:0 | — | 001 | R/W | 保留，写 0 |

SLEEP （睡眠模式控制寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | — | 0 | R0 | 没用 |
| 6 | XOSC_STB | 0 | R | 低速时钟状态<br>0 没有打开或者不稳定<br>1 打开且稳定 |
| 5 | HFRC_STB | 0 | R | 主时钟状态<br>0 没有打开或者不稳定<br>1 打开且稳定 |
| 4:3 | RST[1:0] | XX | R | 最后一次复位指示<br>00 上电复位<br>01 外部复位<br>10 看门狗复位 |
| 2 | OSC_PD | 0 | R/W H0 | 节能控制，OSC 状态改变的时候硬件清 0<br>0 不关闭无用时钟<br>1 关闭无用时钟 |
| 1:0 | MODE[1:0] | 0 | R/W | 功能模式选择<br>00 PM0<br>01 PM1<br>10 PM2<br>11 PM3 |

PERCFG（外设控制寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | — | 0 | R0 | 未用 |
| 6 | T1CFG | 0 | R/W | T1 I/O 位置选择<br>0 位置 1<br>1 位置 2 |
| 5 | T3CFG | 0 | R/W | T3 I/O 位置选择<br>0 位置 1<br>1 位置 2 |
| 4 | T4CFG | 0 | R/W | T4 I/O 位置选择<br>0 位置 1<br>1 位置 2 |
| 3:2 | — | 00 | R0 | 未用 |
| 1 | U1CFG | 0 | R/W | 串口 1 位置选择<br>0 位置 1<br>1 位置 2 |
| 0 | U0CFG | 0 | R/W | 串口 0 位置选择 |

| | | | | |
|---|---|---|---|---|
| 5 | | | | 0 位置 1<br>1 位置 2 |

U0CSR（串口 0 控制&状态寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | MODE | 0 | R/W | 串口模式选择<br>0 SPI 模式<br>1 UART 模式 |
| 6 | RE | 0 | R/W | 接收使能<br>0 关闭接收<br>1 允许接收 |
| 5 | SLAVE | 0 | R/W | SPI 主从选择<br>0 SPI 主<br>1 SPI 从 |
| 4 | FE | 0 | R/W0 | 串口帧错误状态<br>0 没有帧错误<br>1 出现帧错误 |
| 3 | ERR | 0 | R/W0 | 串口校验结果<br>0 没有校验错误<br>1 字节校验出错 |
| 2 | RX_BYTE | 0 | R/W0 | 接收状态<br>0 没有接收到数据<br>1 接收到一字节数据 |
| 1 | TX_BYTE | 0 | R/W0 | 发送状态<br>0 没有发送<br>1 最后一次写入 U0BUF 的数据已经发送 |
| 0 | ACTIVE | 0 | R | 串口忙标志<br>0 串口闲<br>1 串口忙 |

U0GCR （串口 0 常规控制寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | CPOL | 0 | R/W | SPI 时钟极性<br>0 低电平空闲<br>1 高电平空闲 |
| 6 | CPHA | 0 | R/W | SPI 时钟相位<br>0 由 CPOL 跳向非 CPOL 时采样，由非 CPOL 跳向 CPOL 时输出<br>1 由非 CPOL 跳向 CPOL 时采样，由 CPOL 跳向非 CPOL 时输出 |
| 5 | ORDER | 0 | R/W | 传输位序 |

| | | | | |
|---|---|---|---|---|
| | | | | 0 低位在先<br>1 高位在先 |
| 4:0 | BAUD_E[4:0] | 0x00 | R/W | 波特率指数值，与 BAUD_F 决定波特率 |

U0BAUD （串口 0 波特率控制寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | BAUD_M[7:0] | 0X00 | R/W | 波特率尾数，与 BAUD_E 决定波特率 |

U0BUF（串口 0 收发缓冲器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | DATA[7:0] | 0X00 | R/W | UART0 收发寄存器 |

ADCCON1

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | EOC | 0 | R H0 | ADC 结束标志位<br>0 ADC 进行中<br>1 ADC 转换结束 |
| 6 | ST | 0 | R W1 | 手动启动 AD 转换（读 1 表示当前正在进行 AD 转换）<br>0 没有转换<br>1 启动 AD 转换（STSEL=11） |
| 5:4 | STSEL[1:0] | 11 | R/W | AD 转换启动方式选择<br>00 外部触发<br>01 全速转换，不需要触发<br>10 T1 通道 0 比较触发<br>11 手工触发 |
| 3:2 | RCTRL[1:0] | 00 | R/W | 16 位随机数发生器控制位（写 01，10 会在执行后返回 00）<br>00 普通模式（13x 打开）<br>01 开启 LFSR 时钟一次<br>10 生成调节器种子<br>11 信用随机数发生器 |
| 1:0 | - | 11 | R/W | 保留，总是写设置为 1 |

ADCCON3

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7:6 | SREF[1:0] | 00 | 读/写 | 选择单次 AD 转换参考电压<br>00 内部 1.25V 电压 |

| | | | | |
|---|---|---|---|---|
| | | | | 01 外部参考电压 AIN7 输入<br>10 模拟电源电压<br>11 外部参考电压 AIN6-AIN7 输入 |
| 5:4 | SDIV[1:0] | 01 | 读/写 | 选择单次 A/D 转换分辨率<br>00 8 位 (64dec)<br>01 10 位 (128dec)<br>10 12 位 (256dec)<br>11 14 位 (512dec) |
| 3:0 | SCH[3:0] | 00 | 读/写 | 单次 A/D 转换选择，如果写入时 ADC 正在运行，则在完成序列 A/D 转换后立刻开始，否则写入后立即开始 A/D 转换，转换完成后自动清 0 |
| | | | | 0000 AIN0<br>0001 AIN1<br>0010 AIN2<br>0011 AIN3<br>0100 AIN4<br>0101 AIN5<br>0110 AIN6<br>0111 AIN7<br>1000 AIN0- AIN1<br>1001 AIN2- AIN3<br>1010 AIN4- AIN5<br>1011 AIN6- AIN7<br>1100 GND<br>1101 正电源参考电压<br>1110 温度传感器<br>1111 1/3 模拟电压 |

## 1.10.3、实验相关函数

写在程序中的子函数及功能列写如下：
void Delay(uint n);
定性延时，参见实验一

void initUARTtest(void);
函数原型：

```
void initUARTtest(void)
{
```

```
    CLKCON &= ~0x40;                    //晶振
    while(!(SLEEP & 0x40));         //等待晶振稳定
    CLKCON &= ~0x47;                    //TICHSPD128 分频，CLKSPD 不分频
    SLEEP |= 0x04;                      //关闭不用的 RC 振荡器

    PERCFG = 0x00;                  //位置 1 P0 口
    P0SEL = 0x3c;                   //P0 用作串口

    U0CSR |= 0x80;                  //UART 方式
    U0GCR |= 10;                    //baud_e = 10;
    U0BAUD |= 216;                  //波特率设为 57600
    UTX0IF = 1;

    U0CSR |= 0X40;                  //允许接收
    IEN0 |= 0x84;                   //开总中断，接收中断
}
```

函数功能：将 I/0 P10,P11 设置为输出去控制 LED， 将系统时钟设为高速晶振，将 P0 口设置为串口 0 功能引脚，串口 0 使用 UART 模式，波特率设为 57600，允许接收。在使用串口之前调用。

void UartTX_Send_String(char *Data,int len)
函数原型：

```
void UartTX_Send_String(char *Data,int len)
{
  int j;
  for(j=0;j<len;j++)
  {
    U0DBUF = *Data++;
    while(UTX0IF == 0);
    UTX0IF = 0;
  }
}
```

函数功能：串口发送数据， *data 为发送缓冲的指针，len 为发送数据的长度，在初始化串口后才可以正常调用。

void initTempSensor(void)；
函数原型：

```
void initTempSensor(void){
   DISABLE_ALL_INTERRUPTS();


   SET_MAIN_CLOCK_SOURCE(0);


   *((BYTE __xdata*) 0xDF26) = 0x80;
}
```

函数功能：将系统时钟设为晶振，设 AD 目标为片机温度传感器。

INT8 getTemperature(void)；

函数原型：

```
INT8 getTemperature(void){
   UINT8     i;
   UINT16    accValue;
   UINT16    value;

   accValue = 0;
   for( i = 0; i < 4; i++ )
   {
      ADC_SINGLE_CONVERSION(ADC_REF_1_25_V | ADC_14_BIT | ADC_TEMP_SENS);
      ADC_SAMPLE_SINGLE();
      while(!ADC_SAMPLE_READY());

      value =    ADCL >> 2;
      value |= (((UINT16)ADCH) << 6);

      accValue += value;
   }
   value = accValue >> 2; // devide by 4

   return ADC14_TO_CELSIUS(value);
}
```

函数功能：连续进行 4 次 AD 转换，将得到的结果求均值后将 AD 结果转换为温度返回。

## 1.10.3、重要的宏定义

将片内温度传感器 AD 转换的结果转换成温度。

```
#define ADC14_TO_CELSIUS(ADC_VALUE)       ( ((ADC_VALUE) >> 4) - 315)
```

# 1.11、CC2430 基础实验十一 1/3AVDD

## 1.11.1、实验介绍

将 AD 的源设为 1/3 电源电压，并将转换得到温度通过串口送至电脑。

## 1.11.2、实验相关寄存器

实验中操作了的寄存器有 CLKCON,SLEEP,PERCFG,U0CSR,U0GCR,U0BAUD,IEN0,U0DUB,ADCCON1,ADCCON3,ADCH,ADCL,等寄存器。

**CLKCON** 参见实验十
**SLEEP** 参见实验十
**PERCFG** 参见实验十
**U0CSR** 参见实验十
**U0GCR** 参见实验十
**U0BAUD** 参见实验十
**U0BUF** 参见实验十
**ADCCON1** 参见实验十
**ADCCON3** 参见实验十
**IEN0** 参见实验五

## 1.11.3、 实验相关函数

写在程序中的子函数及功能列写如下：
void Delay(uint n)；
定性延时，参见实验一

void initUARTtest(void)；参见实验十

void UartTX_Send_String(char *Data,int len)；参见实验十

void InitialAD(void)；
函数原型：

```
void InitialAD(void)
{
        //P1 out
        P1DIR = 0x03;           //P1 控制 LED
        led1 = 1;
        led2 = 1;           //关 LED
```

```
    ADCH &= 0X00;              //清 EOC 标志
    ADCCON3=0xbf;              //单次转换,参考电压为电源电压，对 1/3 AVDD 进行 A/D 转换
                               //14 位分辨率
    ADCCON1 = 0X30;            //停止 A/D

    ADCCON1 |= 0X40;   //启动 A/D;
}
```

函数功能：设 P10,P11 设为输出控制 LED 灯，将 AD 转换源设为电源电压，ADC 结果分辨率设为 14 位（最高精度），AD 模式为单次转换，启动 ADC 转换。

# 1.12、CC2430 基础实验十二 AVDD

## 1.12.1、实验介绍

将 AD 的源设为电源电压，AD 参考电压为 AVDD，并将转换得到温度通过串口送至电脑。

## 1.12.1、实验相关寄存器

实验中操作了的寄存器有 CLKCON,SLEEP,PERCFG,U0CSR,U0GCR,U0BAUD,IEN0,U0DUB,ADCCON1,ADCCON3,ADCH,ADCL,等寄存器。

**CLKCON** 参见实验十
**SLEEP** 参见实验十
**PERCFG** 参见实验十
**U0CSR** 参见实验十
**U0GCR** 参见实验十
**U0BAUD** 参见实验十
**U0BUF** 参见实验十
**ADCCON1** 参见实验十
**ADCCON3** 参见实验十
**IEN0** 参见实验五

## 1.12.2、实验相关函数

写在程序中的子函数及功能列写如下：
void Delay(uint n)；
定性延时，参见实验一

void initUARTtest(void)；参见实验十

void UartTX_Send_String(char *Data,int len)；参见实验十

void InitialAD(void)；
函数原型：

```
void InitialAD(void)
{
	//P1 out
	P1DIR = 0x03;		//P1 控制 LED
	led1 = 1;
	led2 = 1;		//关 LED

	ADCH &= 0X00;			//清 EOC 标志
	ADCCON3=0xbf;			//单次转换,参考电压为电源电压，对 1/3 AVDD 进行 A/D 转换
					//14 位分辨率
	ADCCON1 = 0X30;			//停止 A/D

	ADCCON1 |= 0X40;	//启动 A/D
}
```

函数功能：设 P10,P11 设为输出控制 LED 灯，将 AD 转换源设为电源电压，参考电压为电源电压，ADC 结果分辨率设为 14 位（最高精度），AD 模式为单次转换，启动 ADC 转换。

# 1.13、CC2430 基础实验十三 串口发数

## 1.13.1、实验介绍

从 CC2430 上通过串口不断地发送字串“UART0 TX Test”。实验使用 CC2430 的串口 1，波特率为 57600。

## 1.13.2、实验相关寄存器

实验中操作了的寄存器有:
P1,P1DIR,CLKCON,SLEEP,PERCFG,U0CSR,U0GCR,U0BAUD,IEN0,U0DUB,等寄存器。

**CLKCON** 参见实验十
**SLEEP** 参见实验十
**PERCFG** 参见实验十
**U0CSR** 参见实验十
**U0GCR** 参见实验十
**U0BAUD** 参见实验十

**U0BUF** 参见实验十
**IEN0** 参见实验五

## 1.13.3、实验相关函数

写在程序中的子函数及功能列写如下：
void Delay(uint n)；
定性延时，参见实验一

void initUARTtest(void)；
函数原型：

```
void initUARTtest(void)
{

    CLKCON &= ~0x40;                //晶振
    while(!(SLEEP & 0x40));         //等待晶振稳定
    CLKCON &= ~0x47;                //TICHSPD128 分频，CLKSPD 不分频
    SLEEP |= 0x04;          //关闭不用的 RC 振荡器

    PERCFG = 0x00;              //位置 1 P0 口
    P0SEL = 0x3c;               //P0 用作串口
    P2DIR &= ~0XC0;                         //P0 优先作为串口 0

    U0CSR |= 0x80;              //UART 方式
    U0GCR |= 10;                //baud_e
    U0BAUD |= 216;              //波特率设为 57600
    UTX0IF = 0;
}
```

函数功能：初始化串口 0，将 I/O 映射到 P0 口，P0 优先作为串口 0 使用，UART 工作方式，波特率为 57600。使用晶振作为系统时钟源。

void UartTX_Send_String(char *Data,int len)；
函数原型：

```
void UartTX_Send_String(char *Data,int len)
{
  int j;
  for(j=0;j<len;j++)
  {
    U0DBUF = *Data++;
    while(UTX0IF == 0);
    UTX0IF = 0;
  }
}
```

函数功能：串口发字串，*Data 为发送缓存指针，len 为发送字串的长度，只能是在初始化

函数 void initUARTtest(void)之后调用才有效。发送完毕后返回，无返回值。

# 1.14、CC2430 基础实验十四　在 PC 用串口控制 LED

## 1.14.1、实验介绍

在 PC 上从串口向 CC2430 发数，即可控制 LED 灯的亮灭，控制数据的格式为“灯编号 开｜关 #”，红色 LED 编号为 1，绿色 LED 编号为 2，0 是关灯，1 是开灯，如打开红色 LED 的命令是“11#”。

## 1.14.2、实验相关寄存器

实验中操作了的寄存器有:
P1,P1DIR,P1SEL,CLKCON,SLEEP,PERCFG,U0CSR,U0GCR,U0BAUD,IEN0,U0DUB, 等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**CLKCON** 参见实验十
**SLEEP** 参见实验十
**PERCFG** 参见实验十
**U0CSR** 参见实验十
**U0GCR** 参见实验十
**U0BAUD** 参见实验十
**U0BUF** 参见实验十

## 1.14.3、实验相关函数

写在程序中的子函数及功能列写如下：
void Delay(uint n)；定性延时，参见实验一

void initUARTtest(void)；
函数原型：

```
void initUARTtest(void)
{

    CLKCON &= ~0x40;                    //晶振
    while(!(SLEEP & 0x40));        //等待晶振稳定
    CLKCON &= ~0x47;                   //TICHSPD128 分频，CLKSPD 不分频
    SLEEP |= 0x04;          //关闭不用的 RC 振荡器
```

```
    PERCFG = 0x00;              //位置 1 P0 口
    P0SEL = 0x3c;               //P0 用作串口
    P2DIR &= ~0XC0;                             //P0 优先作为串口 0

    U0CSR |= 0x80;              //UART 方式
    U0GCR |= 10;                //baud_e
    U0BAUD |= 216;              //波特率设为 57600
    UTX0IF = 0;
}
```

函数功能：初始化串口 0，将 I/O 映射到 P0 口，P0 优先作为串口 0 使用，UART 工作方式，波特率为 57600。使用晶振作为系统时钟源。

void UartTX_Send_String(char *Data,int len)；
函数原型：

```
void UartTX_Send_String(char *Data,int len)
{
  int j;
  for(j=0;j<len;j++)
  {
    U0DBUF = *Data++;
    while(UTX0IF == 0);
    UTX0IF = 0;
  }
}
```

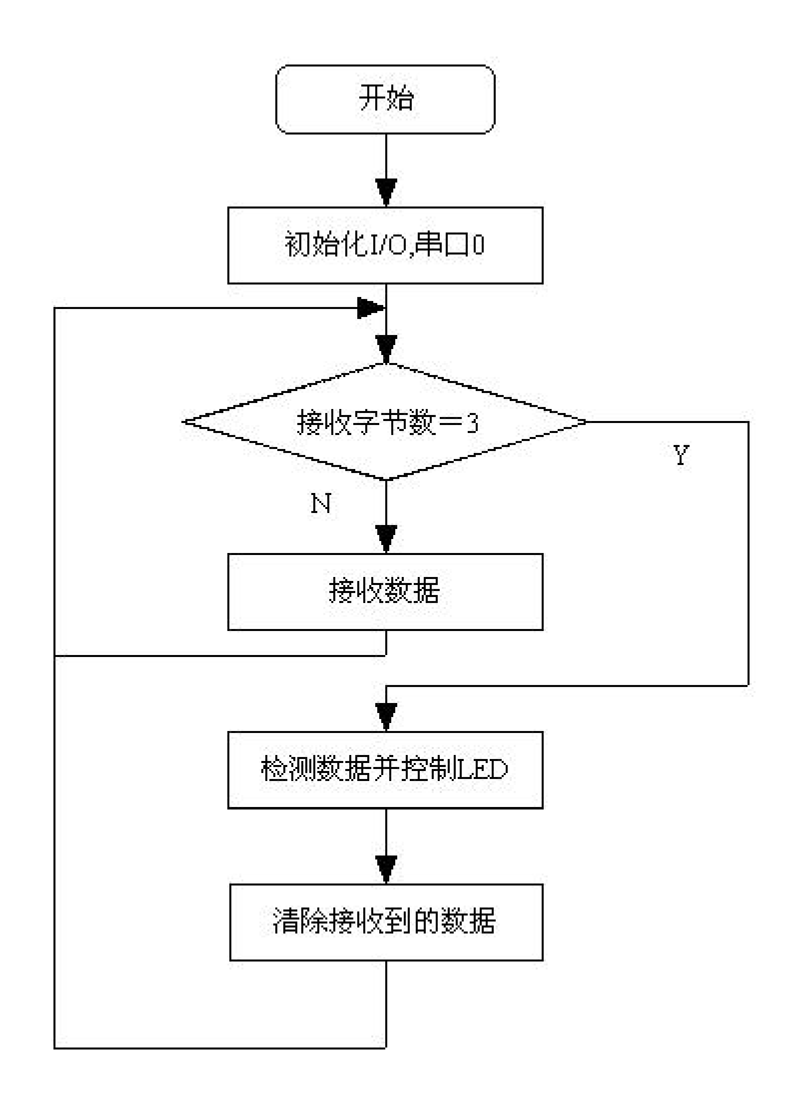
函数功能：串口发字串，*Data 为发送缓存指针，len 为发送字串的长度，只能是在初始化函数 void initUARTtest(void)之后调用才有效。发送完毕后返回，无返回值。

void UART0_ISR(void)；
函数原型：

```
__interrupt void UART0_ISR(void)
 {
     URX0IF = 0;                //清中断标志
     temp = U0DBUF;
 }
```

函数功能：一旦有数据从串口送到 CC2430，则立即进入中断，进入中断后将接收的数据先存放到 temp 变量，然后在主程序中去处理接收到的数据。

## 1.14.4、实验流程图




# 1.15、CC2430 基础实验十五　在 PC 用串口收数并发数

## 1.15.1、实验介绍

在 PC 上从串口向 CC2430 发任意长度为 30 字节的字串，若长度不足 30 字节，则以 "# "为字串末字节，CC2430 在收到字节后会将这一字串从串口反向发向 PC,用串口助手可以显示出来。

## 1.15.2、实验相关寄存器

实验中操作了的寄存器有:
P1,P1DIR,P1SEL,CLKCON,SLEEP,PERCFG,U0CSR,U0GCR,U0BAUD,IEN0,U0DUB, 等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**CLKCON** 参见实验十
**SLEEP** 参见实验十
**PERCFG** 参见实验十
**U0CSR** 参见实验十
**U0GCR** 参见实验十
**U0BAUD** 参见实验十
**U0BUF** 参见实验十

## 1.15.3、实验相关函数

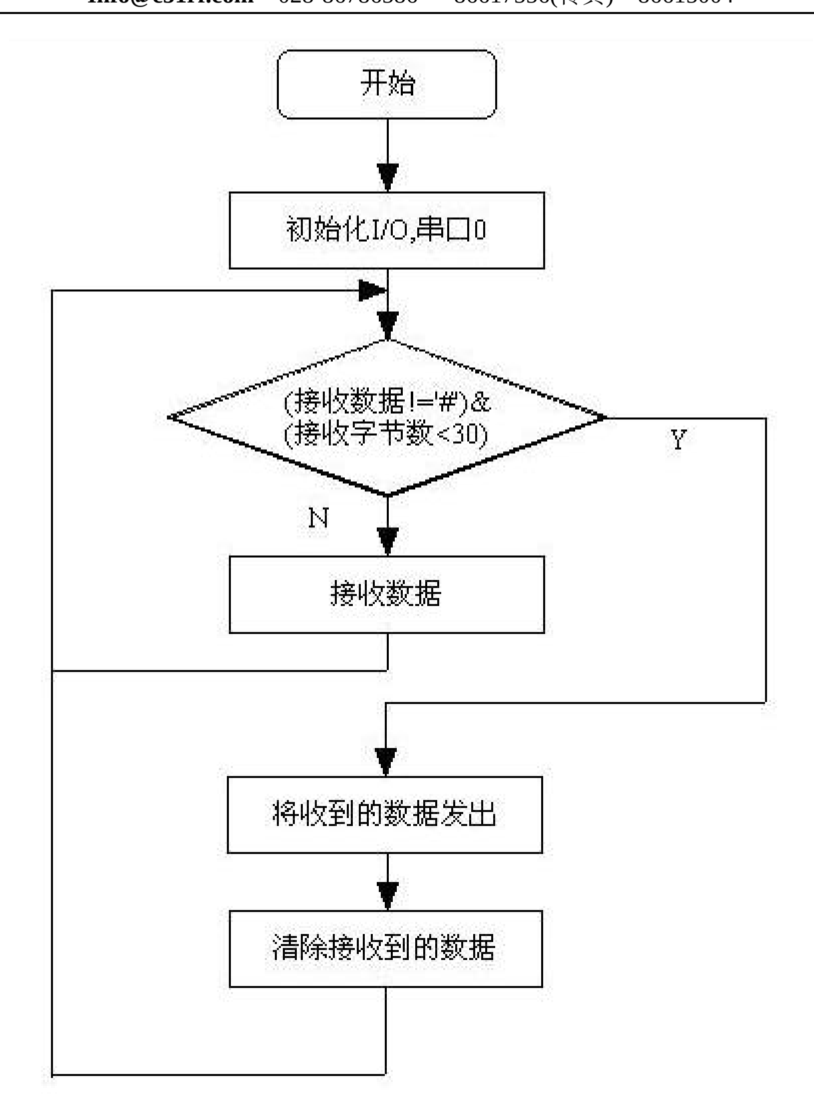
写在程序中的子函数及功能列写如下：
void Delay(uint n)；定性延时，参见实验一

void initUARTtest(void)；参见实验十五

void UartTX_Send_String(char *Data,int len)；参见实验十五

void UART0_ISR(void)；参见实验十五

## 1.15.4、实验流程图




# 1.16、CC2430 基础实验十六　串口时钟 PC 显示

## 1.16.1、实验介绍

利用 CC2430 的定时器 1 产生秒信号，

## 1.16.2、实验相关寄存器

实验中操作了的寄存器有:
P1,P1DIR,P1SEL,T1CTL,T1CCTL0,T1CC0H,T1CC0L,IEN0,IEN1,CLKCON,SLEEP,PERCFG,

U0CSR,U0GCR,U0BAUD,IEN0,U0DUB,等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**T1CTL** 参见实验四
**IEN0** 参见实验十
**CLKCON** 参见实验十
**SLEEP** 参见实验十
**PERCFG** 参见实验十
**U0CSR** 参见实验十
**U0GCR** 参见实验十
**U0BAUD** 参见实验十
**U0BUF** 参见实验十

IEN1（中断使能寄存器 1）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:6 | - | 00 | R0 | 没有，读出为 0 |
| 5 | P0IE | 0 | R/W | P0 口中断使能<br>0 关中断<br>1 开中断 |
| 4 | T4IE | 0 | R/W | 定时器 4 中断使能<br>0 关中断<br>1 开中断 |
| 3 | T3IE | 0 | R/W | 定时器 3 中断使能<br>0 关中断<br>1 开中断 |
| 2 | T2IE | 0 | R/W | 定时器 2 中断使能<br>0 关中断<br>1 开中断 |
| 1 | T1IE | 0 | R/W | 定时器 1 中断使能<br>0 关中断<br>1 开中断 |
| 0 | DMAIE | 0 | R/W | DMA 传输中断使能<br>0 关中断<br>1 开中断 |

T1CCTL0（T1 通道 0 捕获/比较寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | CPSEL | 0 | R/W | T1 通道 0 捕捉设定<br>0 捕捉引脚输入<br>1 捕捉 RF 中断 |
| 6 | IM | 1 | R/W | T1 通道 0 中断掩码 |

| | | | | |
|---|---|---|---|---|
| | | | | 0 关中断<br>1 开中断 |
| 5:3 | CMP[2:0] | 000 | R/W | T1 通道 0 模式比较输出选择，指定计数值过 T3CC0 时的发生事件<br>000 输出置 1（发生比较时）<br>001 输出清 0（发生比较时）<br>010 输出翻转<br>011 输出置 1（发生上比较时）输出清 0（计数值为 0 或 UP/DOWN 模式下发生下比较）<br>100 输出清 0（发生上比较时）输出置 1（计数值为 0 或 UP/DOWN 模式下发生下比较）<br>101 没用<br>110 没用<br>111 没用 |
| 2 | MODE | 0 | R/W | T1 通道 0 模式选择<br>0 捕获<br>1 比较 |
| 1 | CPM[1:0] | 00 | R/W | T1 通道 0 捕获模式选择<br>00 没有捕获<br>01 上升沿捕获<br>10 下降沿捕获<br>11 边沿捕获 |

T1CC0H（T1 通道 0 捕获值/比较值高字节寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | T1CC0[15:8] | 0X00 | R/W | T1 通道 0 捕获值/比较值高字节 |

T1CC0L（T1 通道 0 捕获值/比较值低字节寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | T1CC0[7:0] | 0X00 | R/W | T1 通道 0 捕获值/比较值低字节 |

## 1.16.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(uint n)；定性延时，参见**实验一**

void initUARTtest(void)；参见**实验**十五

void UartTX_Send_String(char *Data,int len)；参见**实验**十五

void UART0_ISR(void)；参见**实验**十五

void InitT1(void)；

函数原型：

```
void InitT1(void)
{
   T1CCTL0 = 0X44;
   //T1CCTL0 (0xE5)
   //T1 ch0  中断使能
   //比较模式

   T1CC0H = 0x03;
   T1CC0L = 0xe8;
   //0x0400 = 1000D)

   T1CTL |= 0X02;
   //start count
   //在这里没有分频。
   //使用比较模式  MODE = 10(B)

   IEN1 |= 0X02;
   IEN0 |= 0X80;
   //开 T1 中断
}
```

函数功能：开 T1 中断，T1 为比较计数模式。因 T1 计数时钟为 0.25M（见 void InitClock(void) 说明），T1CC0 = 0X03E8 ＝ 1000，因此 250 次中断溢出为 1s。

void InitClock(void)；

函数原型：

```
void InitClock(void)
{
   CLKCON = 0X38;
   //TICKSPD = 111  定时器计数时钟源  0.25M
   while(!(SLEEP&0X40));
   //等晶振稳定
```

```
}
```

函数功能：设置系统时钟为晶振，同时将计数器时钟设为 0.25M。晶振振荡稳定后退出函数。

void InitUART0(void)；

函数原型：

```
void InitUART0(void)
{
        PERCFG = 0x00;                    //位置 1 P0 口
        P0SEL = 0x3c;                     //P0 用作串口

        U0CSR |= 0x80;                    //UART 方式
        U0GCR |= 10;                      //baud_e
        U0BAUD |= 216;                    //波特率设为 57600
        UTX0IF = 1;

        U0CSR |= 0X40;                    //允许接收
        IEN0 |= 0x84;                     //开总中断，接收中断
}
```

函数功能：串口 0 映射位置 1，UART 方式，波特率 57600，允许接收，开接收中断。

void T1_ISR(void)；

函数原型：

```
__interrupt void T1_ISR(void)
 {
           IRCON &= ~0x02;//清中断标志
           counter++;
           if(counter == 250)
           {
              counter = 0;
              timetemp = 1;              //一秒到
              led1 = ~led1;             //  调试指示用
           }
 }
```

函数功能：T1 中断服务程序，每 250 次中断将 timetemp 置 1，表示 1 秒时间到，同时改变 LED 的状态。

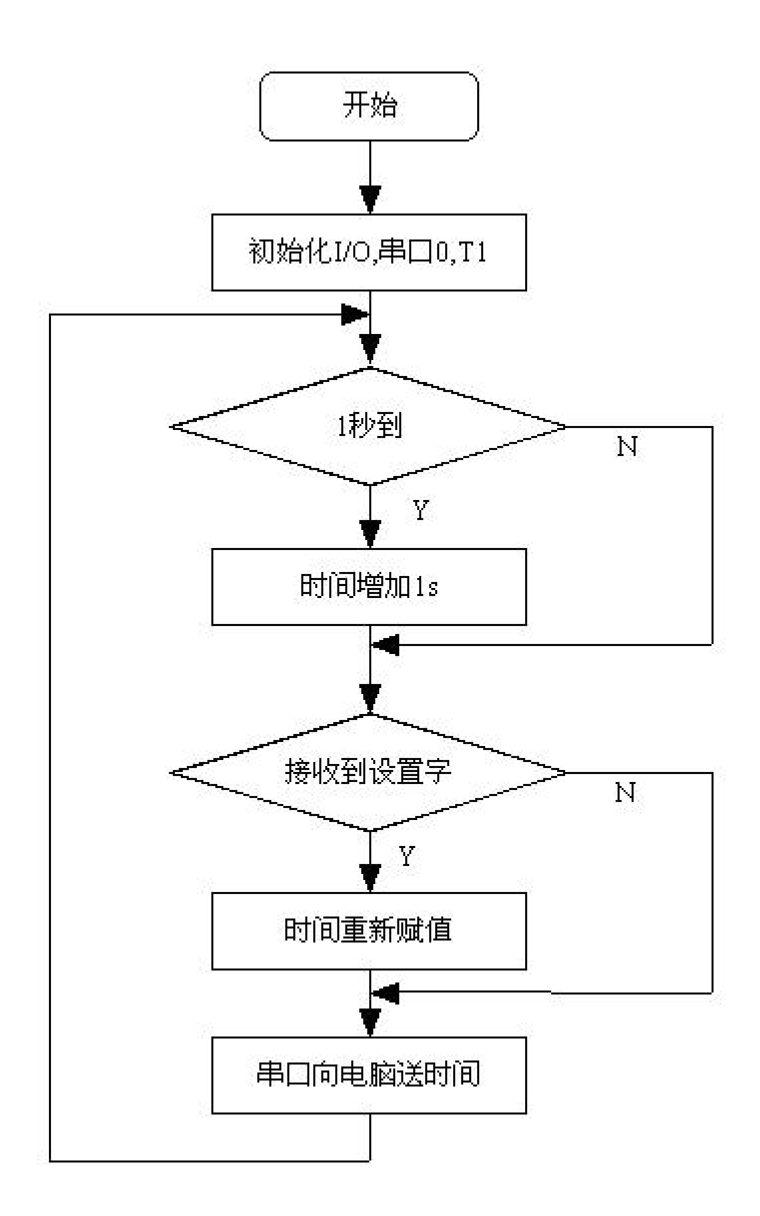
void UART0_ISR(void)；

函数原型：

```
__interrupt void UART0_ISR(void)
 {
        URX0IF = 0;                       //清中断标志
        temp = U0DBUF;
 }
```

函数功能：从串口 0 接收用于设置时间的字符。

## 1.16.4、实验流程图



# 1.17、CC2430 基础实验十七　系统睡眠工作状态

## 1.17.1、实验介绍

在小灯闪烁 10 次以后进入低功耗模式 PM3。CC2430 一共有 4 种功耗模式，分别是PM0,PM1,PM2,PM3，以 PM3 功耗最低。

## 1.17.2、实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR,P1SEL, CLKCON,SLEEP,PCON 等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**CLKCON** 参见实验十
**SLEEP** 参见实验十

PCON（电源模式控制寄存器）

| 位号 | 位名 | 复位值 | 可操作性 | 功能描述 |
|---|---|---|---|---|
| 7:2 | - | 0X00 | R/W | 未用 |
| 1 | - | 0 | R0 | 未用，读出为 0 |
| 0 | IDLE | 0 | R0/W H0 | 电源模式控制，写 1 将进入由 SLEEP.MODE 指定的电源模式，读出一定为 0 |

## 1.17.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(void); 参见实验一
void Initial(void); 参见实验十七

## 1.17.4、重要的宏定义

设置 CC2430 功耗模式，选定后立刻进入相应功耗模式。

```
#define SET_POWER_MODE(mode)                         \
    do {                                             \
        if(mode == 0)          { SLEEP &= ~0x03; } \
        else if (mode == 3)   { SLEEP |= 0x03;    } \
```

```
        else { SLEEP &= ~0x03; SLEEP |= mode;    } \
        PCON |= 0x01;                              \
        asm("NOP");                                \
    }while (0)
```

### 1.17.5、功耗测定方法

将本次实验的程序写入无线生产的 CC2430 模块，将测量电流表串接入 CC2430 模块的供电电路，待小灯信步闪烁后测电流，然后根据 P = U*I 即可得到功率，提示：可以在程序中将小灯关闭，进一步降低功耗。

## 1.18、CC2430 基础实验十八　系统唤醒

### 1.18.1、实验介绍

本次实验使能外部 I/O 中断唤醒 CC2430，每次唤醒红色 LED 闪烁 10 次，然后进入低功耗模式，在进入 PM3 之前程序会将两个 LED 灯关闭。在应用中也可以不关闭以指示 CC2430 处于低功耗模式，可以中断激活。

### 1.18.2、实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR,P1SEL,P1IEN,P1CTL,IEN2,IEN0,P1IFG,P1INP,P2INP,CLKCON,SLEEP 等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**P1IEN** 参见实验九
**PICTL** 参见实验九
**IEN2** 参见实验九
**IEN0** 参见实验五
**P1IFG** 参见实验九
**P1INP** 参见实验二
**CLKCON** 参见实验十
**SLEEP** 参见实验十

P2INP（P2 输入模式寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7 | PDUP2 | 0 | 可读/写 | P2 口上/下拉选择<br>0 上拉 |

| | | | | |
|---|---|---|---|---|
| | | | | 1 下拉 |
| 6 | PDUP1 | 0 | 可读/写 | P1 口上/下拉选择<br>0 上拉<br>1 下拉 |
| 5 | PDUP0 | 0 | 可读/写 | P0 口上/下拉选择<br>0 上拉<br>1 下拉 |
| 4 | MDP2_4 | 0 | 可读/写 | P2_4 输入模式<br>0 上拉<br>1 下拉 |
| 3 | MDP2_3 | 0 | 可读/写 | P2_3 输入模式<br>0 上拉<br>1 下拉 |
| 2 | MDP2_2 | 0 | 可读/写 | P2_2 输入模式<br>0 上拉<br>1 下拉 |
| 1 | MDP2_1 | 0 | 可读/写 | P2_1 输入模式<br>0 上拉<br>1 下拉 |

## 1.18.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(void); 参见实验一

void Init_IO_AND_LED(void);
函数原型：

```
void Init_IO_AND_LED(void)
{
    P1DIR = 0X03;
    RLED = 1;
    YLED = 1;

    P1SEL &= ~0X0C;
    P1DIR &= ~0X0C;
    P1INP   &= ~0X0c;//有上拉、下拉
    P2INP &= ~0X40; //选择上拉

    P1IEN |= 0X0c;     //P12 P13
    PICTL |= 0X02;     //下降沿
```

```
    EA = 1;
    IEN2 |= 0X10;   //P1IE = 1;

    P1IFG |= 0x00;    //P12 P13
};
```

函数功能：置 P10,P11 为输出，打开 P1 口的中断，P1 口下降沿触发中断。

void PowerMode(uchar sel);

函数原型：

```
void PowerMode(uchar sel)
{
    uchar i,j;
    i = sel;
    if(sel<4)
    {
        SLEEP &= 0xfc;
        SLEEP |= i;
        for(j=0;j<4;j++);
        PCON = 0x01;
    }
    else
    {
       PCON = 0x00;
    }
}
```

函数功能：使系统进入 sel 指定的电源模式下，这里的 sel 只能是 0－3 之间的数，程序只能在 CPU 全速运行时执行，也就是说函数中能使系统从全速运行进入 PM0-PM3 而不可以从 PM0-PM3 进入全速运行。

# 1.19、CC2430 基础实验十九　睡眠定时器的使用

## 1.19.1、实验介绍

在小灯快速闪烁 5 次后进入睡眠状态 PM2，在 PM2 下睡眠定时器 SLEEP TIMER（ST）仍然可以正常工作，从 0x000000 到 0xffffff 反复计数，当 ST 计数超过写入 ST[2－0]的 0x000f00 时，系统由中断唤醒，小灯闪烁 5 次后进入 PM2，这样周而复始的唤醒工作然后睡眠。系统睡眠的时间为 8 分 32 秒，这已经是最长睡眠时间。

## 1.19.2、实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR,P1SEL, IEN0,ST2,ST1,ST0,CLKCON,SLEEP 等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**IEN0** 参见实验五
**CLKCON** 参见实验十
**SLEEP** 参见实验十

ST2（睡眠定时器 2）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | ST2[7:0] | 0X00 | R/W | 睡眠定时器计数/比较值[23－16]位。读出为 ST 计数值，写入为比较值。读寄存器应先读 ST0，写寄存器就后写 ST0。 |

ST1（睡眠定时器 1）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | ST1[7:0] | 0X00 | R/W | 睡眠定时器计数/比较值[15－8]位。读出为 ST 计数值，写入为比较值。读寄存器应先读 ST0，写寄存器就后写 ST0。 |

ST0（睡眠定时器 0）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:0 | ST0[7:0] | 0X00 | R/W | 睡眠定时器计数/比较值[7－0]位。读出为 ST 计数值，写入为比较值。读寄存器应先读 ST0，写寄存器就后写 ST0。 |

## 1.19.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(void); 参见实验一

void Init_SLEEP_TIMER(void);

函数原型：

```
void Init_SLEEP_TIMER(void)
{
        ST2 = 0X00;
        ST1 = 0X0f;
        ST0 = 0X00;

    EA = 1;    //开中断
    STIE = 1;
    STIF = 0;
}
```

函数功能：打开睡眠定时器 SLEEP TIMER(ST)中断，设置 ST 的中断发生时间为计数值达到 0x000f00 时。

void LedGlint(void);
函数原型：

```
void LedGlint(void)
{
    uchar jj=10;
    while(jj--)
    {
        RLED = !RLED;
        Delay(10000);
    }
}
```

函数功能：让 LED 闪烁 5 次，无返回值。

void ST_ISR(void)；
函数原型：

```
__interrupt void ST_ISR(void)
  {
      STIF = 0;
}
```

函数功能：睡眠定时器中断服务程序，清中断标志，无其他操作。

## 1.19.4、重要的宏定义

使模块上的可控制

```
#define LED_ENABLE(val)                         \
    do{                                         \
        if(val==1)                              \
            {                                   \
                P1SEL &= ~0X03;                 \
                P1DIR |= 0X03;                  \
```

```
            RLED = 1;                        \
            GLED = 1;                        \
          }                                  \
        else                                 \
        {                                    \
            P1DIR &= ~0X03;                  \
        }                                    \
    }while(0)

#define RLED P1_0
#define GLED P1_1
```

选择系统工作时钟源并关闭不用的时钟源

```
#define SET_MAIN_CLOCK_SOURCE(source) \
    do {                                              \
        if(source) {                                  \
            CLKCON |= 0x40;   /*RC*/                  \
            while(!(SLEEP&0X20));   /*待稳*/           \
            SLEEP |= 0x04;          /*关掉不用的*/ \
        }                                             \
        else {                                        \
            SLEEP &= ~0x04;   /*全开*/                 \
            while(!(SLEEP&0X40));/*待稳*/              \
            asm("NOP");                               \
            CLKCON &= ~0x47; /*晶振*/                  \
            SLEEP |= 0x04;    /*关掉不用的*/           \
        }                                             \
    }while (0)

#define CRYSTAL 0
#define RC 1
```

选择系统低速时钟源

```
#define SET_LOW_CLOCK(source)                                  \
        do{                                                    \
             (source==RC)?(CLKCON |= 0X80):(CLKCON &= ~0X80); \
        }while(0)
```

# 1.20、CC2430 基础实验二十 看门狗模式

## 1.20.1、实验介绍

程序在主程序中没有连续改变小灯的状态，而在开始运行时将其关闭，延时后点亮。实验现象是一只小灯不断闪烁，这是因为程序中启动了看门狗，看门狗时间长度为 1 秒，如果 1 秒内没有复位看门狗的话，系统将复位。系统复位后再次开启看门狗，1 秒后复位。

## 1.20.2、实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR,P1SEL, WDTCL,CLKCON 等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**CLKCON** 参见实验十
**SLEEP** 参见实验十

WDCTL（看门狗定时器控制寄存器）

| 位号 | 位名 | 复位值 | 操作性 | 功能描述 |
|---|---|---|---|---|
| 7:4 | CLR[3:0] | 0000 | R/W | 看门狗复位，先写 0xa 再写 0x5 复位看门狗，两次写入不超过 0.5 个看门狗周期，读出为 0000 |
| 3 | EN | 0 | R/W | 看门狗定时器使能位，在定时器模式下写 0 停止计数，在看门狗模式下写 0 无效<br>0 停止计数<br>1 启动看门狗/开始计数 |
| 2 | MODE | 0 | R/W | 看门狗定时器模式<br>0 看门狗模式<br>1 定时器模式 |
| 1:0 | INT[1:0] | 00 | R/W | 看门狗时间间隔选择<br>00 1 秒<br>01 0.25 秒<br>10 15.625 毫秒<br>11 1.9 毫秒<br>（以 32.768K 时钟计算） |

# 1.20.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(void);
函数原型：

```
void Delay(void)
{
        uint n;
        for(n=50000;n>0;n--);
        for(n=50000;n>0;n--);
        for(n=50000;n>0;n--);
        for(n=50000;n>0;n--);
        for(n=50000;n>0;n--);
        for(n=50000;n>0;n--);
        for(n=50000;n>0;n--);
}
```

函数功能：软件延时 10.94ms。
void Init_IO(void);
函数原型：

```
void Init_IO(void)
{
        P1DIR = 0x03;
        led1 = 1;
        led2 = 1;
}
```

函数功能：将 P10，P11 设置为输出控制 LED。

void Init_Watchdog(void);
函数原型：

```
void Init_Watchdog(void)
{
        WDCTL = 0x00;
        //时间间隔一秒，看门狗模式
        WDCTL |= 0x08;
        //启动看门狗
}
```

函数功能：以看门狗模式启动看门狗定时器，看门狗复位时隔 1 秒。

void Init_Clock(void);
函数原型：

```
void Init_Clock(void)
{
        CLKCON = 0X00;
```

```
}
```

函数功能：将系统时钟设为晶振，低速时钟设为晶振，程序对时钟要求不高，不用等待晶振稳定。

# 1.21、CC2430 基础实验二十一 喂狗

## 1.21.1、实验介绍

本实验与实验二十一都以看门狗为学习目标，在实验二十一学会初始化看门狗，同时也知道了看门狗的作用，本实验着重学习复位看门狗。复位看门狗后小灯不会闪烁。

## 1.21.2、实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR,P1SEL, WDTCL,CLKCON 等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**CLKCON** 参见实验十
**SLEEP** 参见实验十
**WDCTL** 参见实验二十

## 1.21.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(void); 参见实验二十
void Init_IO(void); 参见实验二十
void Init_Watchdog(void); 参见实验二十
void Init_Clock(void); 参见实验二十

void FeetDog(void)；
函数原型：

```
void FeetDog(void)
{
       WDCTL = 0xa0;
       WDCTL = 0x50;
}
```

函数功能：复位看门狗，必须在看门狗时间间隔内调用本函数复位看门狗，系统会被强制复位，此时调用本函数已无意义。

# 1.22、CC2430 基础实验二十二 定时唤醒

## 1.22.1、实验介绍

这个实验利用睡眠定时器工作在多个电源模式下这一特性来实现定时唤醒，最长的唤醒时隔为 8 分 32 秒，而最短的时隔可达 30 余微秒。实验中在设定好唤醒时间后让 CC2430 进入 PM2 模式，在达到指定时间后小灯闪烁，之后再次是设定唤醒时间，进入 PM2，唤醒的循环。

## 1.22.2、实验相关寄存器

实验中操作了的寄存器有 P1,P1DIR,P1SEL, ST2,ST1,ST0,CLKCON,SLEEP 等寄存器。

**P1** 参见实验一
**P1DIR** 参见实验一
**P1SEL** 参见实验一
**CLKCON** 参见实验十
**SLEEP** 参见实验十
**ST2** 参见实验十九
**ST1** 参见实验十九
**ST0** 参见实验十九

## 1.22.3、实验相关函数

写在程序中的子函数及功能列写如下：

void Delay(void); 参见实验一
void LedGlint(void); 参见实验十九

void Init_SLEEP_TIMER(void);
函数原型：

```
void Init_SLEEP_TIMER(void)
{
   EA = 1;   //开中断
```

```
  STIE = 1;
  STIF = 0;
}
```

函数功能： 打开睡眠定时器（ST）的中断，并且将 ST 的中断标志位清零。在使用 ST 时必须于 addToSleepTimer()前调用本函数。

void addToSleepTimer(UINT16 sec)；
函数原型：

```
void addToSleepTimer(UINT16 sec)
{
   UINT32 sleepTimer = 0;

   sleepTimer |= ST0;
   sleepTimer |= (UINT32)ST1 <<  8;
   sleepTimer |= (UINT32)ST2 << 16;

   sleepTimer += ((UINT32)sec * (UINT32)32768);

   ST2 = (UINT8)(sleepTimer >> 16);
   ST1 = (UINT8)(sleepTimer >> 8);
   ST0 = (UINT8) sleepTimer;
}
```

函数功能：设置睡眠时间，在 sec 秒以后由 ST 唤醒 CC2430，在调用这个函数之前必须先调用 Init_SLEEP_TIMER（），否则不能唤醒 CC2430。通常在这个函数以后会出现 SET_POWER_MODE(2) 语句。

# 1. 重要的宏定义

改变系统的电源功耗模式

```
#define SET_POWER_MODE(mode)                        \
   do {                                             \
      if(mode == 0)        { SLEEP &= ~0x03; }      \
      else if (mode == 3)  { SLEEP |= 0x03;  }      \
      else { SLEEP &= ~0x03; SLEEP |= mode;  }      \
      PCON |= 0x01;                                 \
      asm("NOP");                                   \
   }while (0)
```

# 2、点对点通信（SPP）

## 2.1 实验目的：

CC2430 点对点通信

## 2.2 实验设备：

CC2430 模块两块，仿真器 1 台

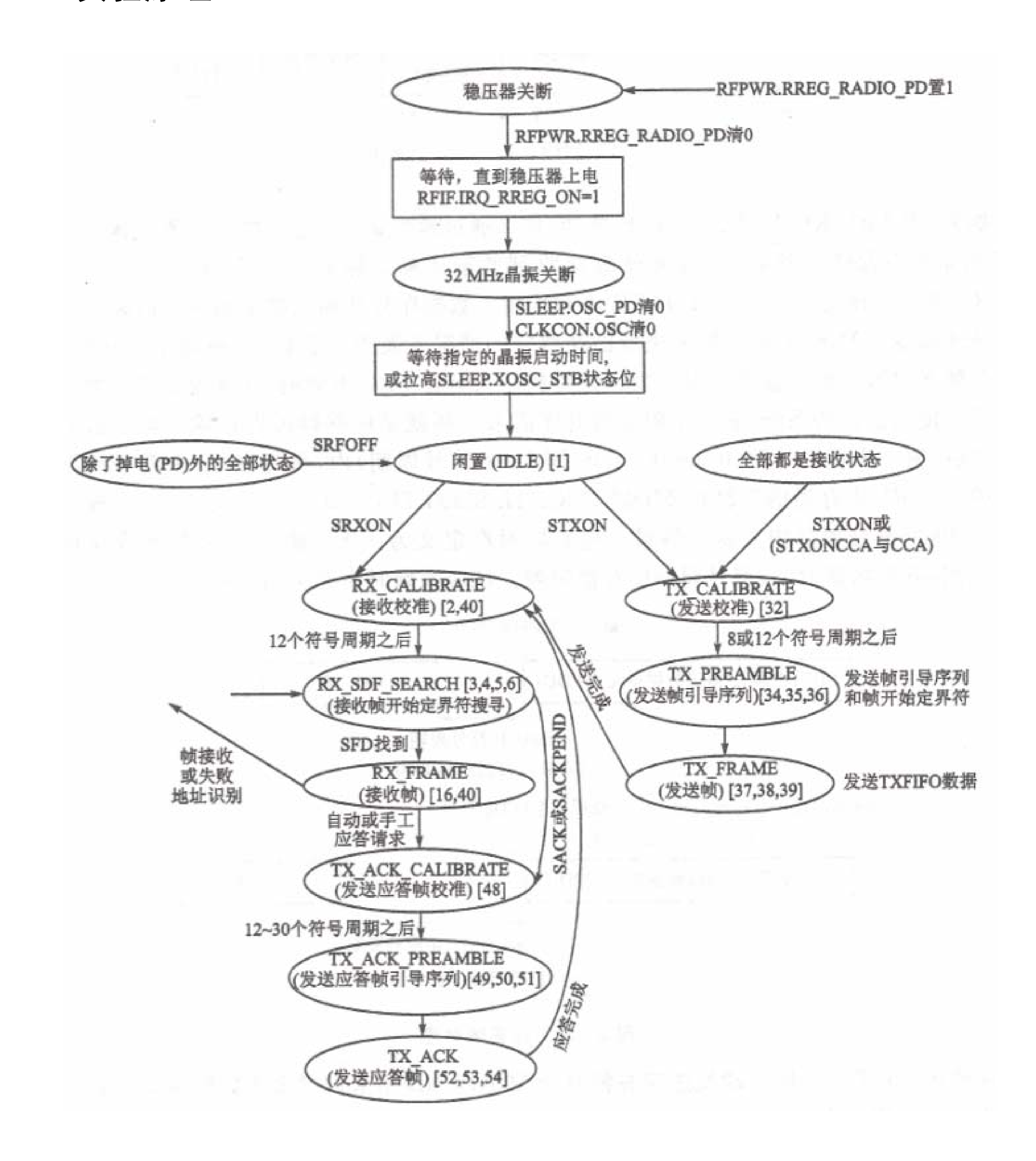
## 2.3 实验内容：

了解 CC2430 点对点通信操作流程

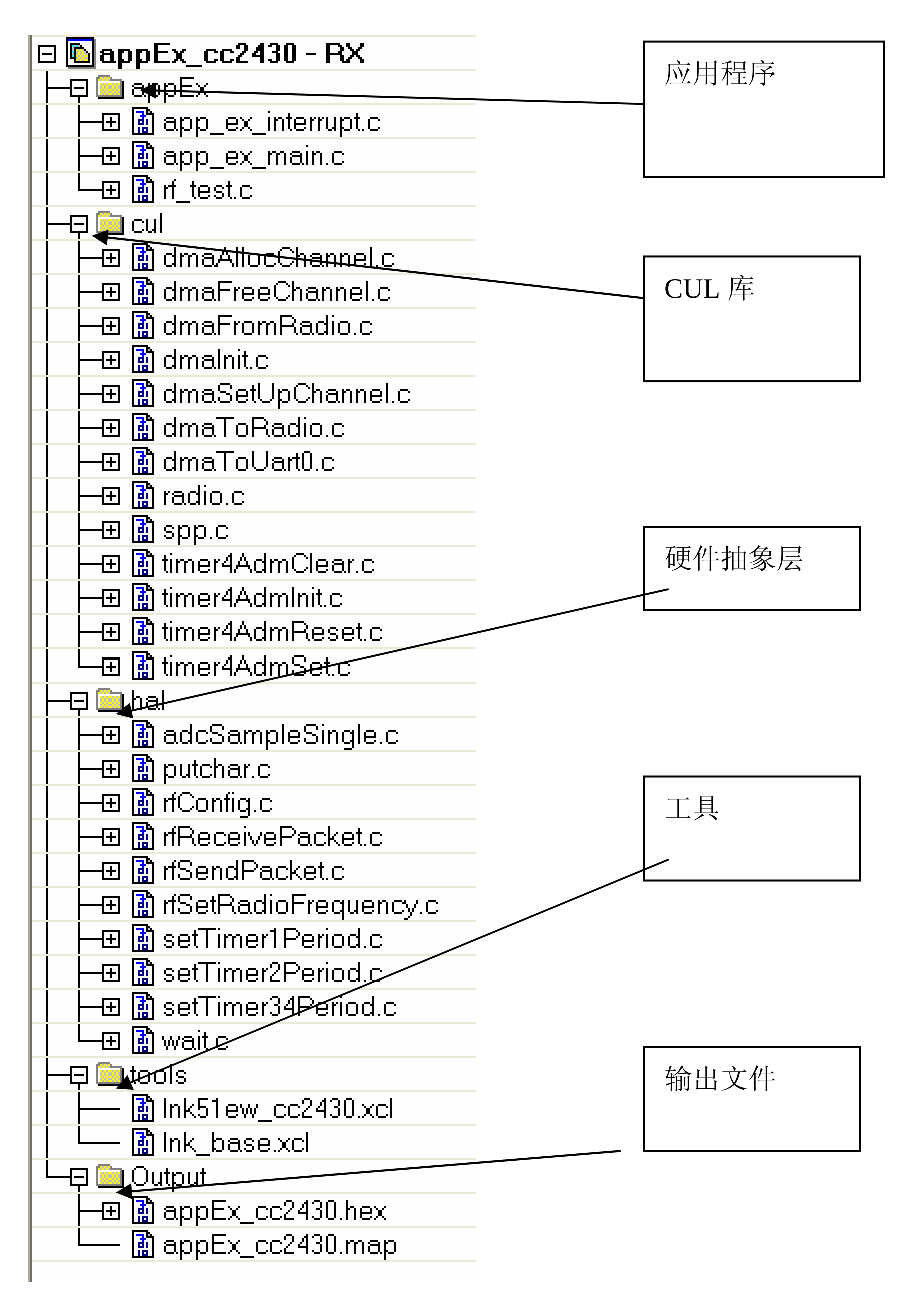
了解 SPP 的使用

了解射频配置

## 2.4 实验原理：


稳压器关断
RFPWR.RREG_RADIO_PD置1
RFPWR.RREG_RADIO_PD清0
等待，直到稳压器上电
RFIF.IRQ_RREG_ON=1
32 MHz晶振关断
SLEEP.OSC_PD清0
CLKCON.OSC清0
等待指定的晶振启动时间，
或拉高SLEEP.XOSC_STB状态位
除了掉电 (PD)外的全部状态
SRFOFF
闲置 (IDLE) [1]
全部都是接收状态
SRXON
STXON
STXON或
(STXONCCA与CCA)
RX_CALIBRATE
(接收校准) [2,40]
TX_CALIBRATE
(发送校准) [32]
12个符号周期之后
8或12个符号周期之后
发送完成
RX_SDF_SEARCH [3,4,5,6]
(接收帧开始定界符搜寻)
TX_PREAMBLE
(发送帧引导序列)[34,35,36]
发送帧引导序列
和帧开始定界符
SFD找到
帧接收
或失败
地址识别
RX_FRAME
(接收帧) [16,40]
TX_FRAME
(发送帧) [37,38,39]
发送TXFIFO数据
SACK或SACKPEND
自动或手工
应答请求
TX_ACK_CALIBRATE
(发送应答帧校准) [48]
12~30个符号周期之后
TX_ACK_PREAMBLE
(发送应答帧引导序列)[49,50,51]
应答完成
TX_ACK
(发送应答帧) [52,53,54]

## 2.5 协议栈目录：




相关文件定义：

# 2.6 几个重要函数

## 2.6.1、射频初始化函数

BOOL sppInit(UINT32 frequency, BYTE address)

功能描述：初始化简单的数据包装协议 Simple Packet Protocol (SPP)，从 DMA 管理器申请两个 DMA 通道，用于分别从 Rx FIFO 和 Tx FIFO 传输数据。定时器 4 管理器同样被设置，这个单元用于在数据包发送后接收器在一定时间内没有返回应答时产生中断。无线部分配置为发送，工作在特定的频率，在发送时自动计算和插入和检查 CRC 值。

参数描述:

UINT32 frequency：RF 的频率（kHz.）;

BYTE address：节点地址

返回：配置成功返回 TRUE，失败返回 FALSE

## 2.6.2、发送数据包函数

BYTE sppSend(SPP_TX_STRUCT* pPacketPointer)

功能描述：发送 length 字节的数据（最多 122），标志，目的地址，源地址在 Tx DMA 通道传送有效载荷到 Tx FIFO 前插入，如果期望应当，设置相应的标志。

参数：

SPP_TX_STRUCT* pPacketPointer：发送数据包头指针

返回：发送成功返回 TRUE，失败返回 FALSE。

## 2.6.3、接收数据

void sppReceive(SPP_RX_STRUCT *pReceiveData)

功能描述：

这个函数使能接收 128 字节，包括头和尾。接收数据通过 DMA 传输到 pReceiveData。DMA 装备同时接收开启。接收数据将触发 DMA，当所有的数据包接收并且移走，DMA 产生一个中断同时运行以前定义的函数 rxCallBack。

参数：

SPP_TX_STRUCT* pPacketPointer：接收数据包头指针

返回：无

## 2.7 程序实现

### 2.7.1、射频初始化应用函数

```
void initRfTest(void)
{
    UINT32 frequency = 2405000;
    INIT_GLED();
    INIT_YLED();
    radioInit(frequency, myAddr);
}
```

这个不用多解释，实际上仅仅调用了 sppInit 函数初始化了射频。

### 2.7.2、发送状态函数

```
void contionuousMode(void)
{
    BOOL res;
    BYTE sendBuffer[] = "Hello";
    while(1)
    {
        GLED = LED_OFF;
        YLED = LED_ON;
        res = radioSend(sendBuffer, sizeof(sendBuffer), remoteAddr, DO_NOT_ACK );
        halWait(200);
        YLED = LED_OFF;
        halWait(200);
        if(res == TRUE)
```

```
            {
                GLED = LED_ON;
                halWait(200);
            }
            else
            {
              GLED = LED_OFF;
              halWait(200);
            }
        }
    }
```

在发送状态调用了 radioSend 函数来发送数据，似乎与前面提到的 sppSend 发送数据不一样，但是实际上是一样的，因为在 radioSend 函数中调用了 sppSend 函数。

## 2.7.3、接收状态

```
void receiveMode(void)
{
    BYTE* receiveBuffer;
    BYTE length;
    BYTE res;
    BYTE sender;
    while(1)
    {
        YLED = LED_ON;
        res = radioReceive(&receiveBuffer, &length, RECEIVE_TIMEOUT, &sender);
        YLED = LED_OFF;

        if(res == TRUE)
        {
```

```
            GLED = LED_ON;
            halWait(200);
        }
        else
        {
            GLED = LED_OFF;
            halWait(200);
        }
        GLED = LED_OFF;
    }
}
```

同样的道理，虽然在这里调用的是 radioReceive 函数，但是在 radioReceive 函数函数里调用了 sppReceive 函数来接收数据。

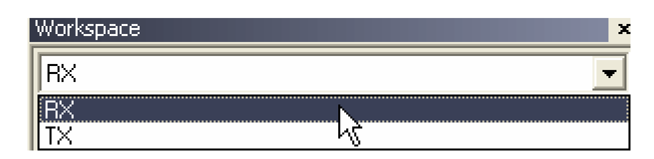
## 2.7.4、射频主函数

```
#ifdef COMPLETE_APPLICATION
void rf_test_main(void){
#else
void main(void){
#endif
    INT_GLOBAL_ENABLE(INT_ON);
    #ifdef RX
    {
            myAddr = ADDRESS_0;
            remoteAddr = ADDRESS_1;
            initRfTest();
                receiveMode();
    }
    #else
```

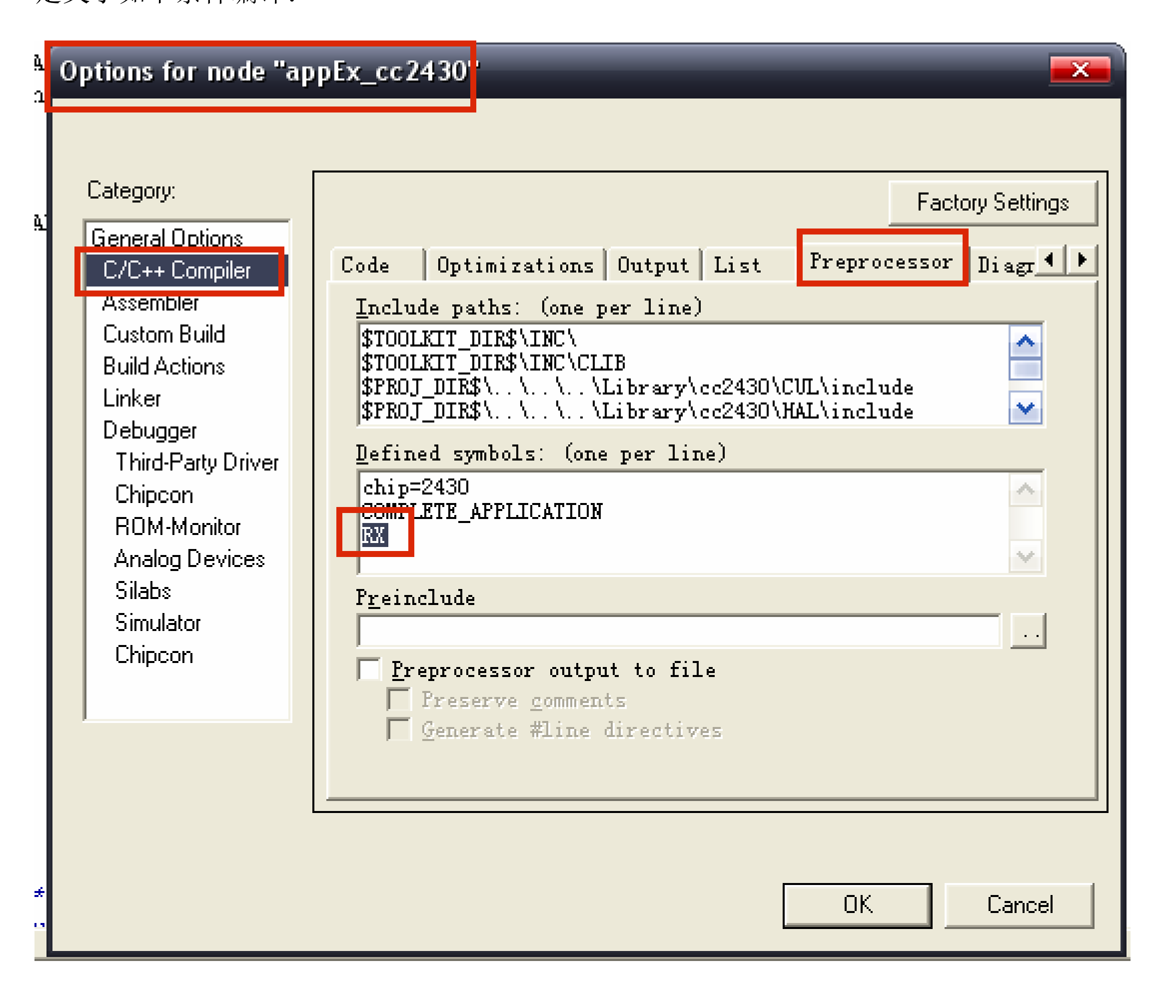
```
    {
                myAddr = ADDRESS_1;
                remoteAddr = ADDRESS_0;
                initRfTest();
                        contionuousMode();
    }
    #endif
}
```

注意这里的几个条件编译，主要是为了在同一个文件下同时编写 TX 和 RX 程序。

例如：在 RX 工程下：



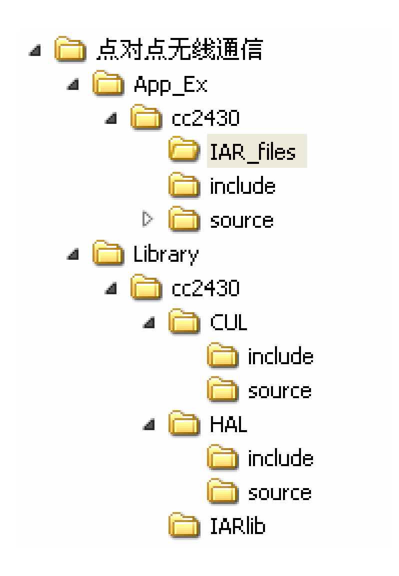

定义了如下条件编译：




表运行接收状态。另一个 TX 就是发送状态。

# 2.8 实验步骤

## 2.8.1、熟悉 SPP 协议



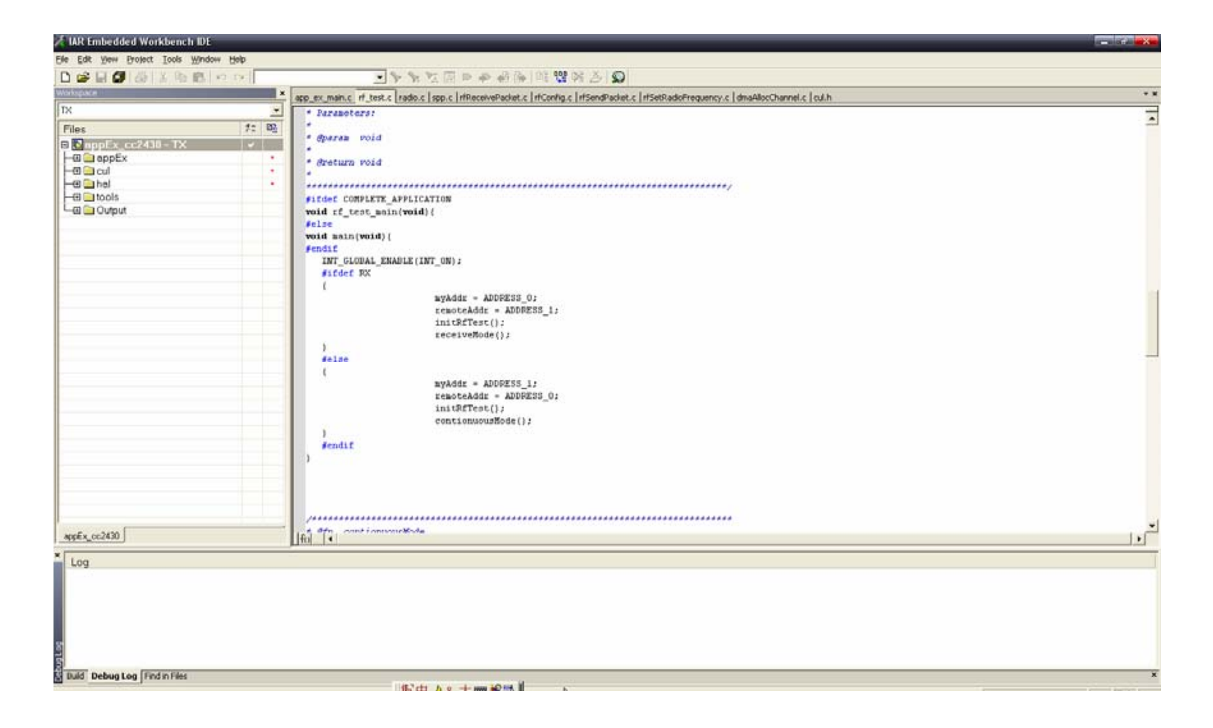

## 2.8.2、工程路径：

……………..点对点无线通信\App_Ex\cc2430\IAR_files

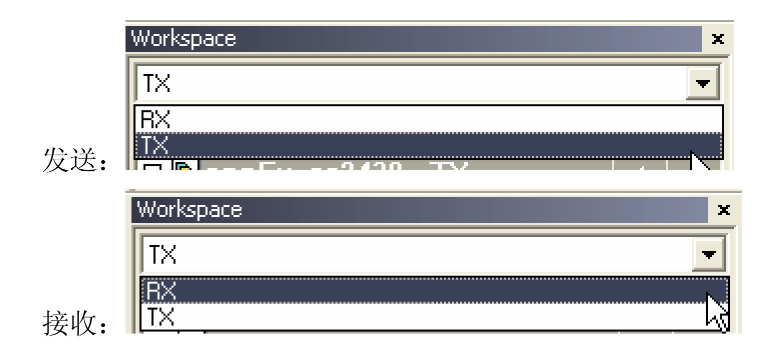
## 2.8.3、打开工程



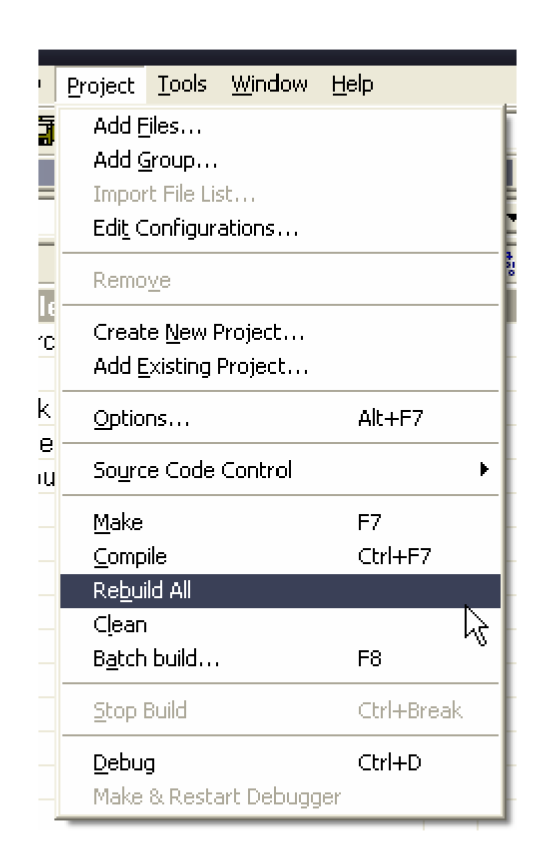
## 2.8.4、工程界面



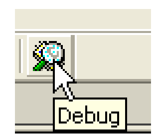

## 2.8.5、选择 RF 状态



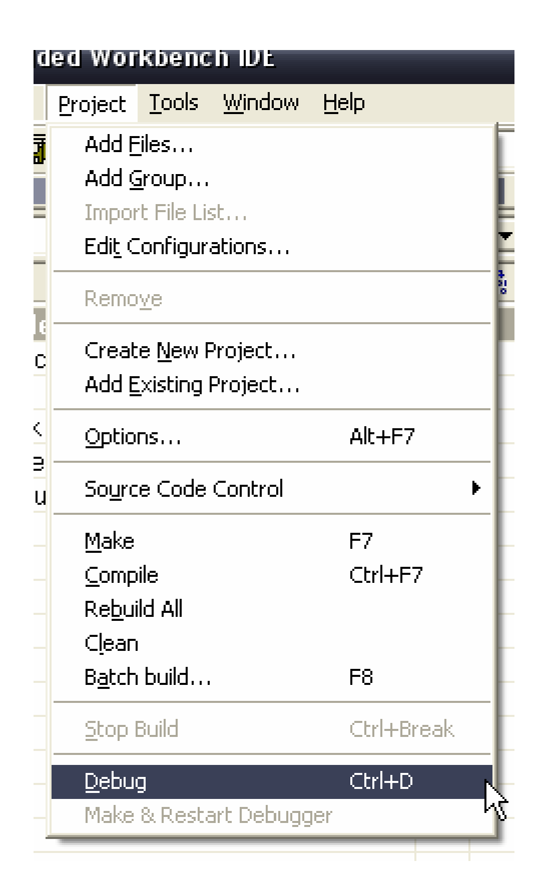
## 2.8.6、编译


Project
Tools
Window
Help
Add Files…
Add Group…
Import File List…
Edit Configurations…
Remove
Create New Project…
Add Existing Project…
Options… Alt+F7
Source Code Control
Make F7
Compile Ctrl+F7
Rebuild All
Clean
Batch build… F8
Stop Build Ctrl+Break
Debug Ctrl+D
Make & Restart Debugger

## 2.8.7、下载程序




或



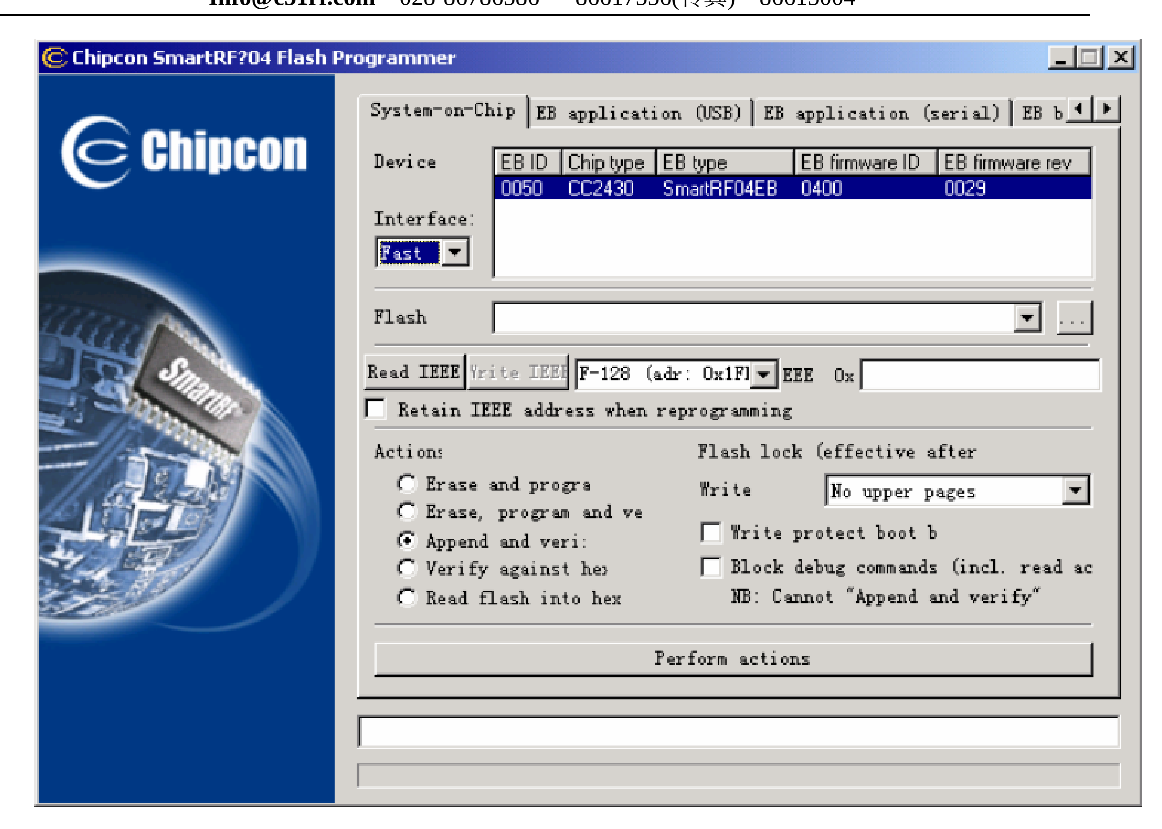

分别将 TX 和 RX 下载到两个模块。

## 2.8.8、64 位物理地址设定。

解压缩 ChipconFlashProgrammer_1_38.zip 文件可以看到 

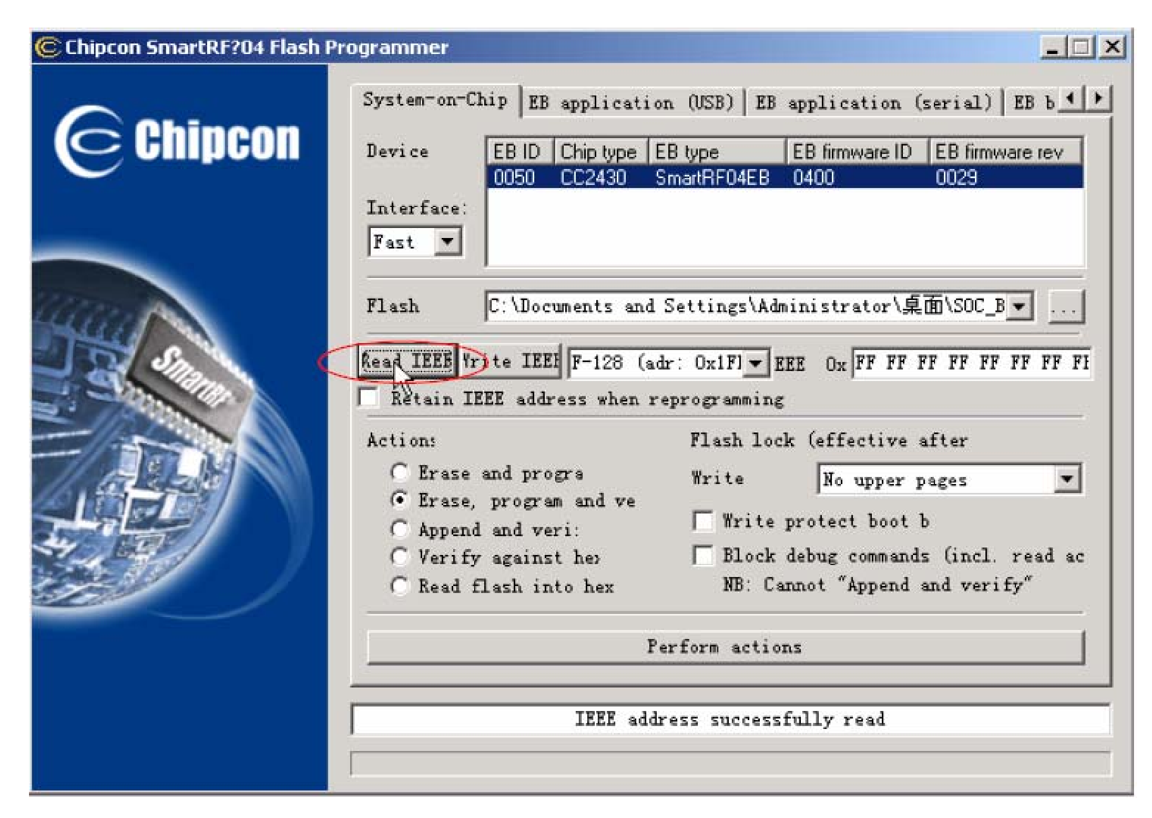
图标，双击这个图标打开 FLASH 烧写工具。

连接上 C51RF-3 型仿真器后再把模块接到仿真器可以看到如下显示：

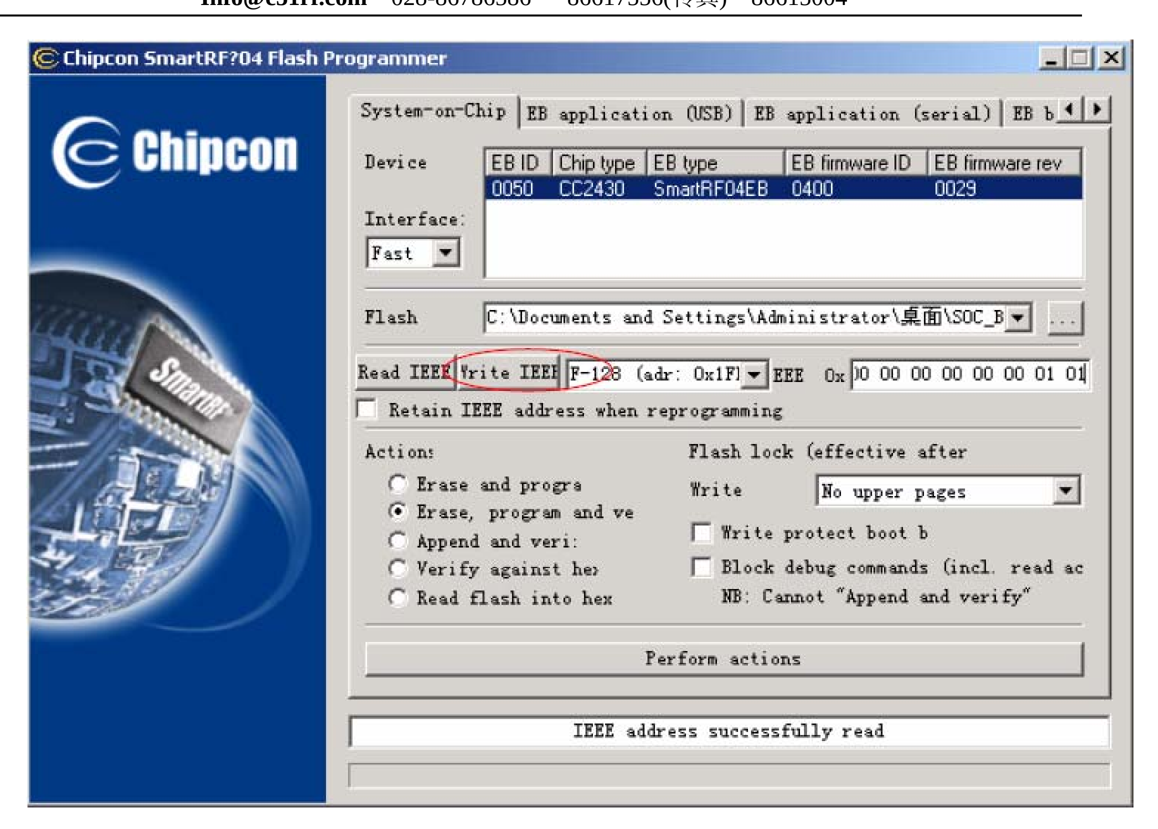



此时可以看到 FLASH 烧写工具已经检测到 CC2430 模块，如果此时没有检测到模块可以按仿真器的复位按键以便检测到模块。

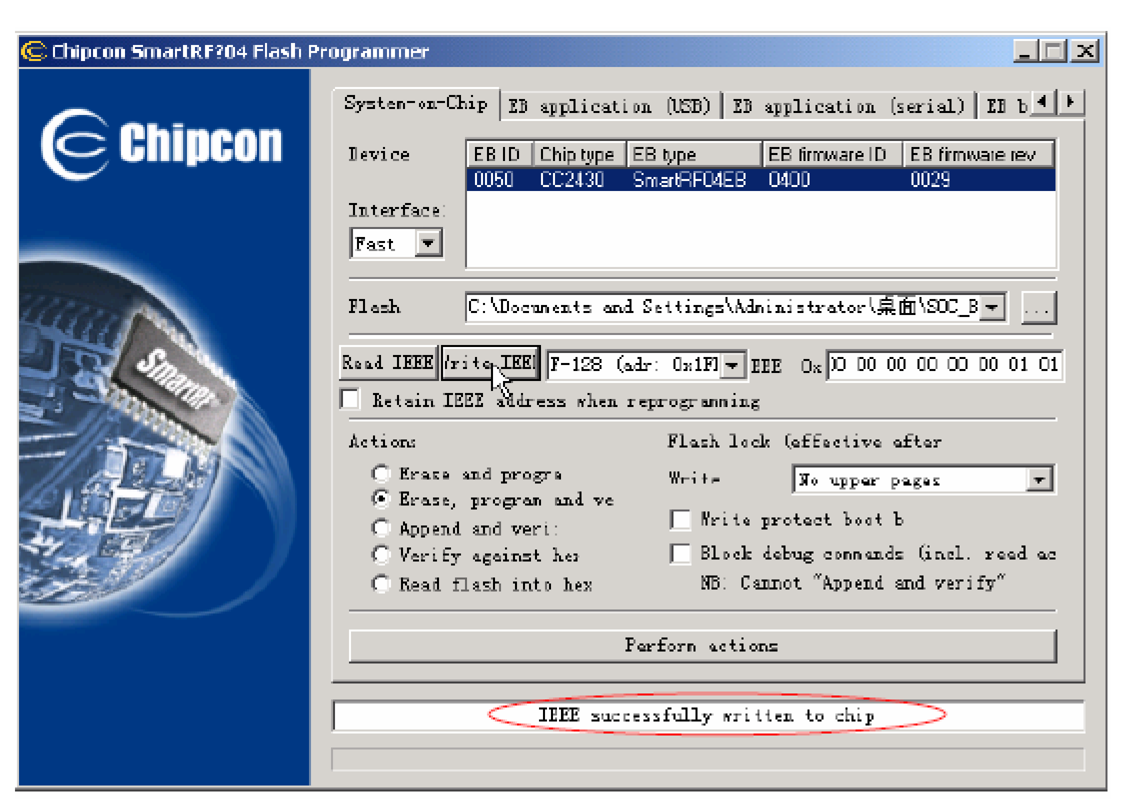
此时点击“Read IEEE”按键，可以读出模块的 64 位物理地址，如图：




然后把物理地址修改为你所要的地址：




最后点击“Write IEEE”按钮，写入64 位物理地址：




此时工具提示“IEEE successfully written to chip”表示地址写入成功。

## 2.9、实验结果

运行的结果是发送和接收模块上的小灯交替闪烁。

从程序上也能分析出：

- 发送和接收正正常工作黄灯闪烁；
- 发送成功发送模块的红灯闪烁；
- 接收成功接收模块上的红灯闪烁。

备注：可能不同的 CC2430 模块灯的颜色不一样，请自行判断。

# 3、点对点无线通信_uart

待补充：………………………………………………….

# 4、点对多点通信-FDMA

了解点对多点的定义，FDMA 原理，熟悉其应用方法及范畴，学会利用 FDMA 原理在 CC2430 实验平台上加以体现。

## 4.1 实验目的

- 了解 FDMA 原理；
- 学会 FDMA 编程方法及技巧；
- 利用 FDMA 原理在 CC2430 实验平台上实现点对多点通讯。

## 4.2 实验内容

让一台接收机，接收两台工作在不同频道的发送机发送的数据，收到数据后通过通过小灯闪烁表示出来。

## 4.3 实验设备

- C51RF-3-PK/C51RF-3-JKS(三个 CC2430 模块、开发地板、电源、USB 线)

- IAR 集成开发环境
- C51RF-3 仿真器

## 4.4 实验原理

无线通讯与有线连接在诸多重要环节上完全不同，这些环节中的异同导致了他们之间的通信质量的差异：

1、无线链路是通过相同的传输媒介——空气来传播无线电信号；

2、误码率比常规有线系统高几个数量级。由于存在上述差异，RF 链路的可靠性比有线链路低。

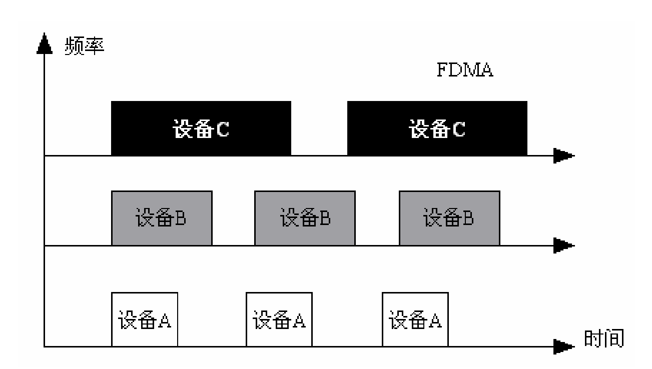
3、为了实现在同一范围内多点间通讯，必须考虑防止数据包在空气中的传输时相互碰撞，为了建立可靠的无线传输通路，必须采用各种方法。例如 TDMA/FDMA/CSMA 等都是无线通讯中常用的办法。

FDMA 是数据通信中的一种技术，即不同的用户分配在时隙相同而频率不同的信道上。按照这种技术，把在频分多路传输系统中集中控制的频段根据要求分配给用户。同固定分配系统相比，频分多址使通道容量可根据要求动态地进行交换。

在 FDMA 系统中，分配给用户一个信道，即一对频谱，一个频谱用作前向信道即基站向移动台方向的信道，另一个则用作反向信道即移动台向基站方向的信道。这种通信系统的基站必须同时发射和接收多个不同频率的信号，任意两个移动用户之间进行通信都必须经过基站的中转，因而必须同时占用 2 个信道(2 对频谱)才能实现双工通信。

以往的模拟通信系统一律采用 FDMA。频分多址(FDMA)是采用调频的多址技术。业务信道在不同的频段分配给不同的用户。如 TACS 系统、AMPS 系统等。频分多址是把通信系统的总频段划分成若干个等间隔的频道(也称信道)分配给不同的用户使用。这些频道互不交叠，其宽度应能传输一路数字话音信息，而在相邻频道之间无明显的串扰。

频分多址 (FDMA)技术将可用的频率带宽拆分为具有较窄带宽的子信道，如图所示。这样每个子信道均独立于其它子信道，从而可被分配给单个发送器。其优点是软件控制上比较简单，其缺陷是子信道之间必须间隔一定距离以防止干扰，频带利用率不高。




## 4.5 FDMA 程序实现

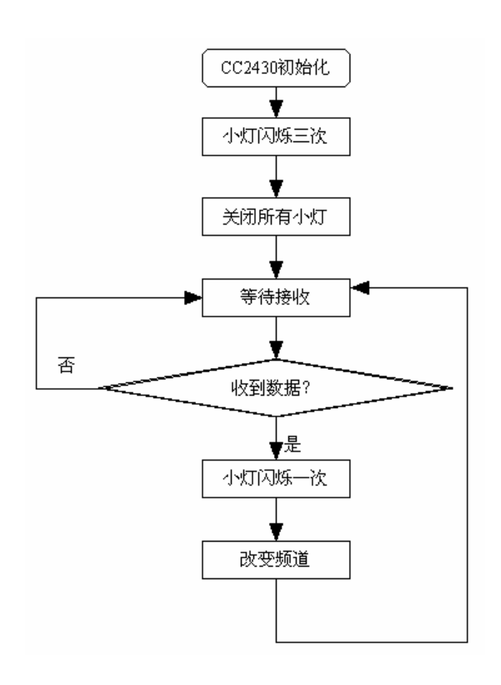
FDMA 是一个多频率的通信方式，在实验中信道的改变是必要的，我们将频段设置在

2.4GHz 上，要改变信道的方法是改变寄存器 **CHANNR**，在改变信道的时候，只需要改变 CHANNR 的值，在下面的代码中给出了两个设备，选择不同信道的方法。

在实验使用三个 CC2430 无线通讯模块，两个发送模块 Tx1 和 Tx2，一个接收模块 Rx。当上电复位后，小灯闪烁三次后，发送模块 1、2 的红黄小灯交替闪烁，并发送数据，发送模块 1 发送的数据为：Connect to NO.1；发送模块 2 发送的数据为：Connect to NO.2。发送模块就会向接收模块发送数据。

模块 Tx1 和模块 Tx2 在编程时，被强制固定在不同的子频道上，模块 Tx1 和模块 Tx2 同时向 Rx 模块发送数据包(因为在不同的子频道上发射，所以在空气中，这些数据包不会发射碰撞，不会出现数据包的传输错误)。而 Rx 模块时时刻刻地扫描监视空气中不同子频道，发现有合格的数据包，就会自动进行接收。这就实现了点(Rx 模块)对多点(模块 Tx1 和模块 Tx2)的可靠无线数据通讯。

## 4.5.1、接收程序流程图




## 4.5.2、接收程序实现

FDMA 接收程序主要是在两个频道上循环监听，如果有收到发送模块的信号，就跳到另外一个频道继续监听。

程序首先是初始化程序，初始化射频部分 CC2420 和内部 CPU。然后红色 LED 点亮后进入接收状态。然后程序进入主循环部分，等待接收信号，如果接收到数据后，相应的小灯会闪烁一次，然后，改变自己的频道等待接收下一个频道的数据。

### 4.5.3、发送流程图

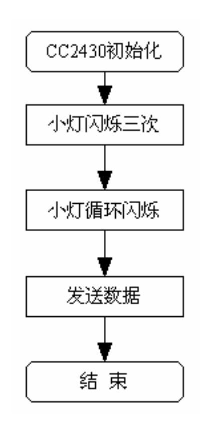



图 5.13 发送流程图

### 4.5.4、发送程序实现

FDMA 发送程序主要功能为循环发送数据，程序开始同样是初始化程序，初始化射频部分 CC2420 和内部 CPU，小灯闪烁三次后，小灯循环闪烁，开始发送数据。

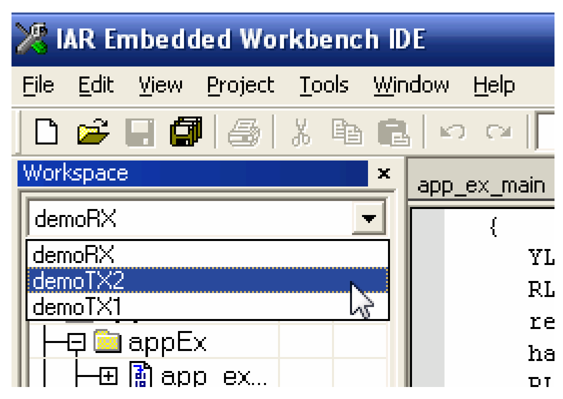
程程序初始化完成之后为了与接收程序的频道对应，我们也根据节点的 ID 号重新设定了节点的发送接收频道，这样使不同的节点各自占有各自唯一的频道。

## 4.6 代码实现（略）

略。（请参考具体源代码，内部有详细的注释，如有疑问请联系我公司技术支持，028-86786586-157/158）

## 4.7 实验步骤

1. 连接好硬件，请参阅系统说明书。
2. 打开 IAR 工程后，可以看到如下图所示的窗口，在本窗口中可以看到，这个工程中共分为三个模块，RX、TX1、TX2。RX 表示发送，TX1\TX2 表示发送，首先选择接收模块。



3. 通过说明书中 IAR 使用方法下载代码，将接收部分代码下载到模块中。运行后小灯闪烁，如果有液晶会有无线龙图表显示和 Receive 字样显示，红、黄小灯同时闪烁三次，然后关闭所有小灯，等待接收数据。
4. 用同样的方法下载 TX2\TX1，程序运行后，液晶同样会显示无线龙图表和模块编号、Transmiting 等字样，小灯同时闪烁三次后，小灯循环闪烁，并发送数据，当两个发送模块都开始发送数据后，就开始通讯了。
5. 构建好以后，接收模块接收数据，可以在液晶上看到接收到的是那个设备的数据。

# 5、ZigBee2004 精简版使用

## 5.1、实验目的：

学会使用 ZigBee2004 精简版协议栈

## 5.2、实验设备：

C51RF-3-CS 或者 C51RF-3-PK

## 5.3、实验内容：

熟悉 ZigBee2004 精简版协议栈结构

学会在 C51RF-3 开发平台上使用 ZigBee2004 精简版协议栈

学会在 ZigBee2004 精简版协议栈添加应用

## 5.4、实验原理：

### 5.4.1、zigbee 介绍

ZigBee 是一种新兴的短距离、低速率、低功耗无线网络技术，主要用于近距离无线连接。

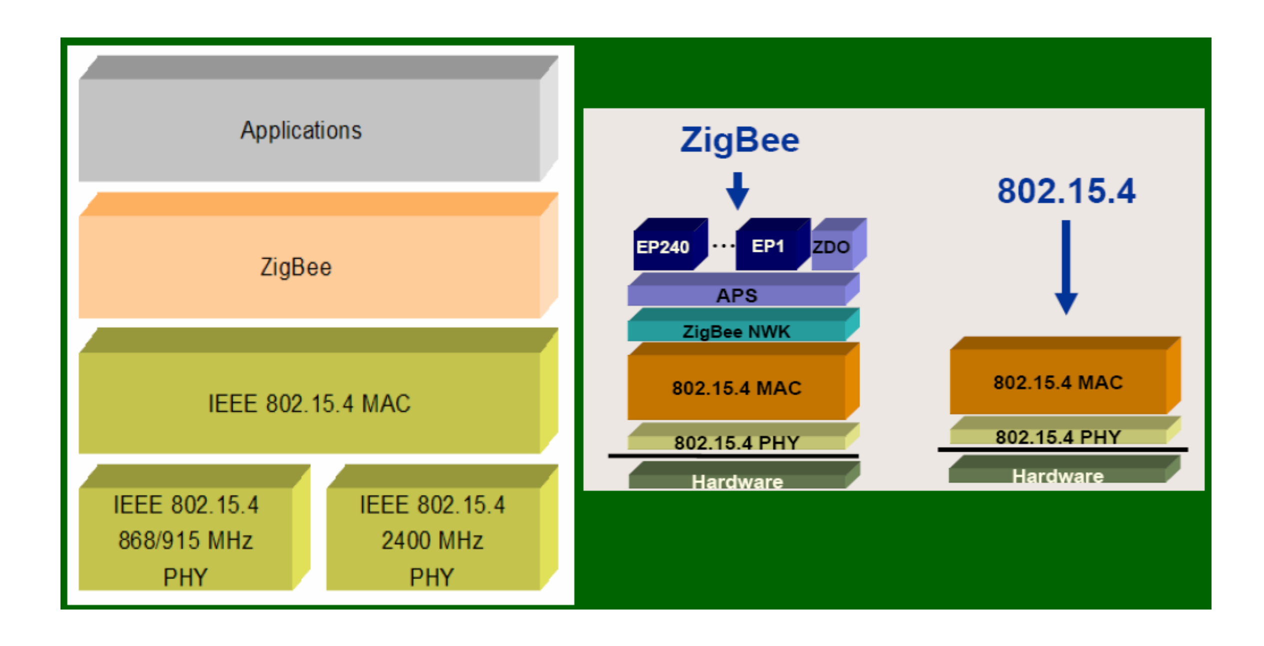
它有自己的无线电标准，在数千个微小的传感器之间相互协调实现通信。这些传感器只需要很低的功耗，以接力的方式通过无线电波将数据从一个传感器传到另一个传感器，因此它们的通信效率非常高。

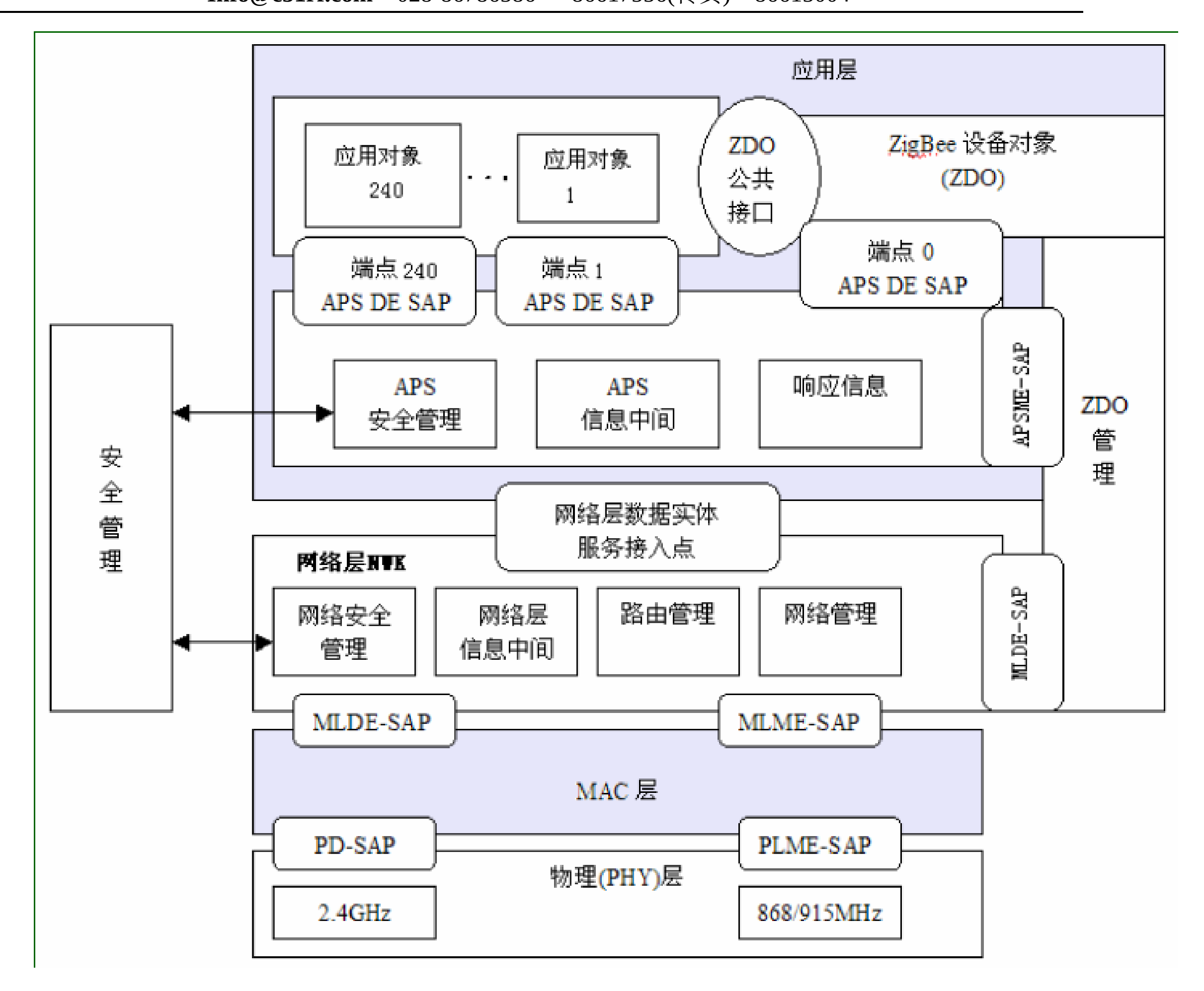
ZigBee 的基础是 IEEE802.15.4，这是 IEEE 无线个人区域网工作组的一项标准，被称作 IEEE802.15.4(ZigBee)技术标准。

ZigBee 不仅只是 802.15.4 的名字。IEEE 仅处理低级 MAC 层和物理层协议，因此 ZigBee 联盟对其网络层协议和 API 进行了标准化。

ZigBee 联盟还开发了安全层。

## 5.4.2、zigbee 结构






## 5.4.3、ZigBee 节点类型

- ZigBee 协调者---coord 为协调者节点
  a) 每个 ZigBee 网络必须有一个
  b) 初始化网络信息.
- ZigBee 路由器---router 为路由节点
  a) 路由信息
- ZigBee 终端节点---rfd 为终端节点
  a) 没有路由功能–低价格

## 5.4.4、ZigBee 物理层

物理层定义了物理无线信道和 MAC 子层之间的接口，提供物理层数据服务和物理层管理服务。 物理层数据服务从无线物理信道上收发数据。 物理管理服务维护一个由物理层

相关数据组成的数据库。

1)ZigBee 的激活；

2)当前信道的能量检测；

3)接收链路服务质量信息；

4)ZigBee 信道接入方式；

5)信道频率选择；

6)数据传输和接收。

## 5.4.5、MAC 层

MAC 层负责处理所有的物理无线信道访问，并产生网络信号、同步信号；支持 PAN 连接和分离，提供两个对等 MAC 实体之间可靠的链路。 MAC 层数据服务：保证 MAC 协议数据单元在物理层数据服务中正确收发。 MAC 层管理服务：维护一个存储 MAC 子层协议状态相关信息的数据库。

网络协调器产生信标；

与信标同步；

支持 PAN(个域网)链路的建立和断开；

为设备的安全性提供支持；

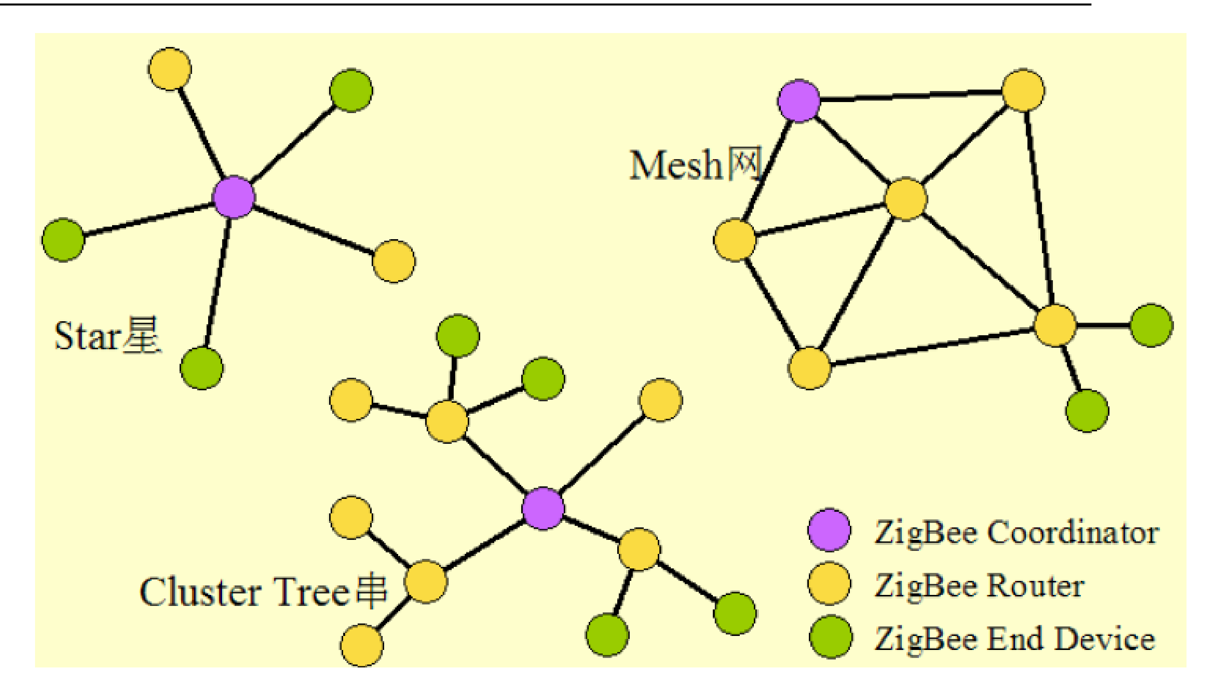
信道接入方式采用免冲突载波检测多址接入(CSMA-CA)机制；

处理和维护保护时隙(GTS)机制；

在两个对等的 MAC 实体之间提供一个可靠的通信链路。

## 5.4.6、网络层

ZigBee 协议栈的核心部分在网络层。网络层主要实现节点加入或离开网络、接收或抛弃其他节点、路由查找及传送数据等功能，支持 Cluster-Tree 等多种路由算法，支持星形（Star）、树形（Cluster-Tree）、网格（Mesh）等多种拓扑结构




网络层功能：

1)网络发现；

2)网络形成；

3)允许设备连接；

4)路由器初始化；

5)设备同网络连接；

6)直接将设备同网络连接；

7)断开网络连接；

8)重新复位设备；

9)接收机同步；

10)信息库维护。

## 5.4.7、应用层

ZigBee 应用层框架包括应用支持层(APS)、ZigBee 设备对象(ZDO)和制造商所定义的应用对象。

应用支持层的功能包括：维持绑定表、在绑定的设备之间传送消息。所谓绑定就是基于两台设备的服务和需求将它们匹配地连接起来。

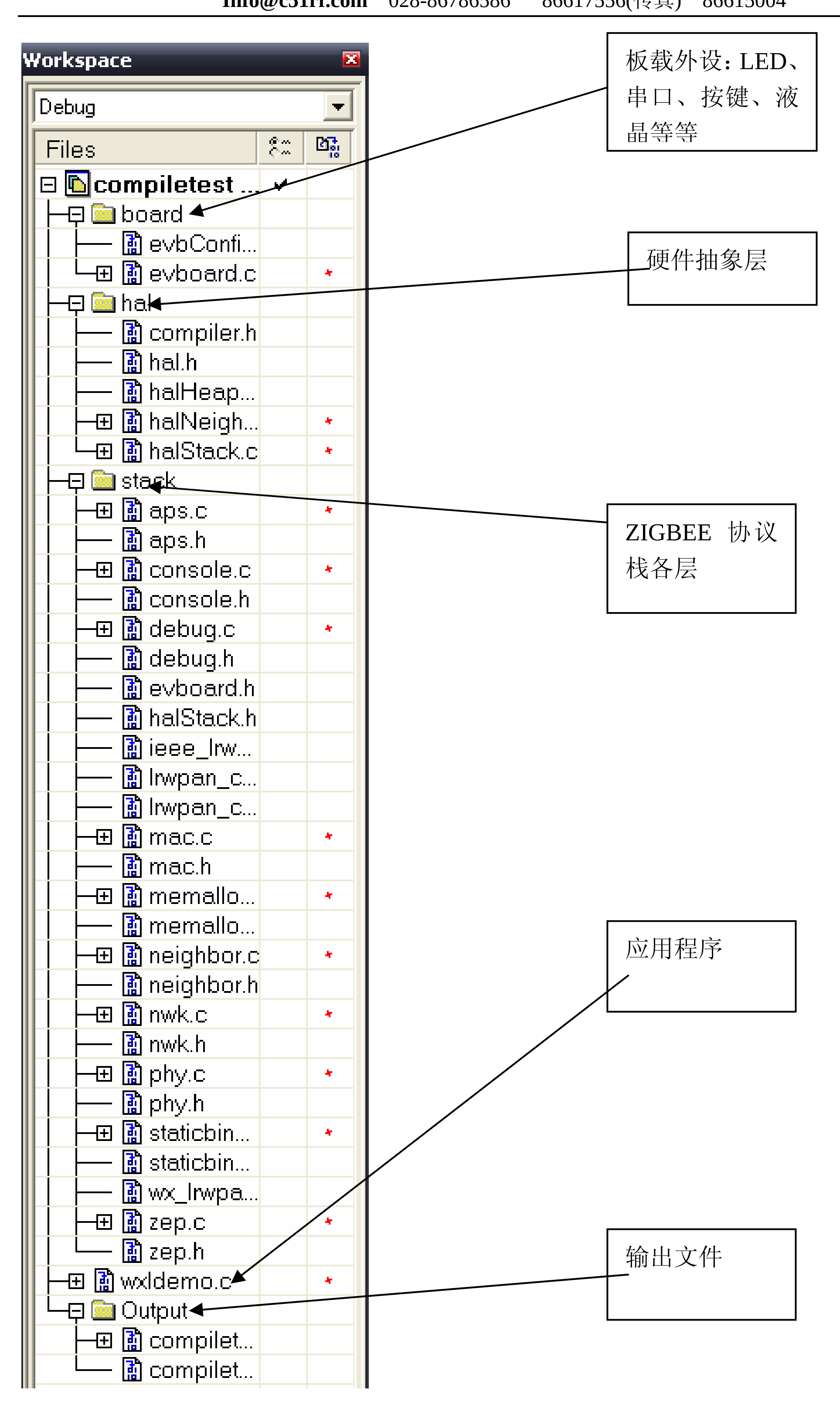
ZigBee 设备对象的功能包括：定义设备在网络中的角色(如 ZigBee 协调器和终端设备)，

发起和响应绑定请求，在网络设备之间建立安全机制。ZigBee 设备对象还负责发现网络中的设备，并且决定向他们提供何种应用服务。 ZigBee 应用层除了提供一些必要函数以及为网络层提供合适的服务接口外，一个重要的功能是应用者可在这层定义自己的应用对象。

## 5.5、协议栈目录：

相关文件定义：

| File | Description |
|---|---|
| | |
| aps.c/h | APS/APL layer |
| nwk.c/h | Network layer |
| mac.c/h | MAC layer |
| phy.c/h | PHY layer |
| neighbor.c/h | Neighbor table, address table functions |
| console.c/h | Console output utility functions |
| debug.c/h | Debugging output utility functions |
| memalloc.c/h | heap management |
| ieee_ pan_defs.h | #defines for 801.15.4 |
| wpan_config.h | #defines for configuring this stack |
| wpan_common_types.h | Common types used by most files |
| staticbind.c/h | functions for accessing static bind table defined in the .h file |
| zep.c/h | Zigbee Zero Endpoint device functions |
| halstack.h, evboard.h | Prototypes for functions providing portability in the HAL, and evaluation board. |


Workspace
Debug
Files
compiletest ...
board
evbConfi...
evboard.c
hal
compiler.h
hal.h
halHeap...
halNeigh...
halStack.c
stack
aps.c
aps.h
console.c
console.h
debug.c
debug.h
evboard.h
halStack.h
ieee_lrw...
lrwpan_c...
lrwpan_c...
mac.c
mac.h
memallo...
memallo...
neighbor.c
neighbor.h
nwk.c
nwk.h
phy.c
phy.h
staticbin...
staticbin...
wx_lrwpa...
zep.c
zep.h
wxldemo.c
Output
compilet...
compilet...
板载外设：LED、串口、按键、液晶等等
硬件抽象层
ZIGBEE 协议栈各层
应用程序
输出文件

# 5.6、实验程序实现：

## 5.6.1、初始化：

```
void main (void){
        //this initialization set our SADDR to 0xFFFF,
        //PANID to the default PANID
        //HalInit, evbInit will have to be called by the user
        numTimeouts = 0;
        my_timer = 0;
        first_packet = TRUE;
        halInit();
        evbInit();   //CC2430芯片初始化

        aplInit();  //init the stack  协议栈初始化
        conPrintConfig();
        ENABLE_GLOBAL_INTERRUPT(); //enable interrupts

        EVB_LED1_OFF();        //LED灯初始化
        EVB_LED2_OFF();

        ping_cnt = 0;
        rxFlag = 0;
        //debug_level = 10;
```

对外设及 CC2430 进行了初始化，并对协议栈也进行了初始化。

## 5.6.2、协调器

```
#ifdef LRWPAN_COORDINATOR        //是协调者
        aplFormNetwork();        //初始化一个新网络
        while(apsBusy()) {apsFSM();} //wait for finish
        conPrintROMString("Network formed, waiting for RX\n");
        EVB_LED2_ON();//LED灯亮
        ppState = PP_STATE_START_RX;
```

可以看出如果是协调器，设备调用了格式化网络函数 aplFormNetwork()。

# 5.6.3、非协调器：

```
#else                                   //是路由或终端
        do {
                aplJoinNetwork();//加入已存在的一个网络
                while(apsBusy()) {apsFSM();} //wait for finish 等待完成加入网络
                if (aplGetStatus() == LRWPAN_STATUS_SUCCESS) {   //成功加入网络
                        EVB_LED2_ON();//LED灯亮
                        //数据输出显示
                        conPrintROMString("Network Join succeeded!\n");
                        conPrintROMString("My ShortAddress is: ");
                        conPrintUINT16(aplGetMyShortAddress());//串口输出显示网络地址
                        conPCRLF();//串口输出换行
                        conPrintROMString("Parent LADDR: ")
                        conPrintLADDR(aplGetParentLongAddress());//串口输出物理地址
                        conPrintROMString(", Parent SADDR: ");
                        conPrintUINT16(aplGetParentShortAddress());
                        conPCRLF();
                        break;
                }else {//加入网络失败
                        conPrintROMString("Network Join FAILED! Waiting, then trying again\n");
                        my_timer= halGetMACTimer();
                        //wait for 2 seconds
                        while ((halMACTimerNowDelta(my_timer))< MSECS_TO_MACTICKS(2*1000));
                }
        } while(1);
```

可以看出如果是非协调器（路由器或终端）那么调用了加入网络函数 aplJoinNetwork()。

**终端（RFD）**

```
#ifdef LRWPAN_RFD    //RFD(终端)节点
        //now send packets
        dstADDR.saddr = 0; //RFD sends to the coordinator   RFD发送数据的目的地址为网络协调器
        ppState = PP_STATE_SEND;
#endif
```

如果是终端设备，那么发送数据的目的地址为 0-----发送数据到协调器。

**路由器（ROU）**

```
#ifdef LRWPAN_ROUTER //路由节点
        //router does nothing, just routes
        DEBUG_PRINTNEIGHBORS(DBG_INFO);
        conPrintROMString("Router, doing its thing.!\n");
        while(1) {apsFSM();}    //应用层处理函数
#endif
```

如果是路由器，不发送数据，只处理应用层数据处理。

## 5.6.4、非路由器

```
#if (defined(LRWPAN_RFD) || defined(LRWPAN_COORDINATOR))    //网络协调器
        //WARNING - this is only for latency testing, max MAC retries is normally
        //set to aMaxFrameRetries (value=3) as defined in mac.h. Setting this to 0 means
        //that there will be no automatic retransmissions of frames if we do not get a MAC ACK back.
        //only do this in your normal code if you want to disable automatic retries
        aplSetMacMaxFrameRetries(0);
        while (1) {
                PingPong();
        }
#endif
```

如果不是路由器，那么调用应用程序 PingPong();

**总结：可以看出，该例子路由器仅仅作为路由功能，只能协调器和终端之间发送数据。**

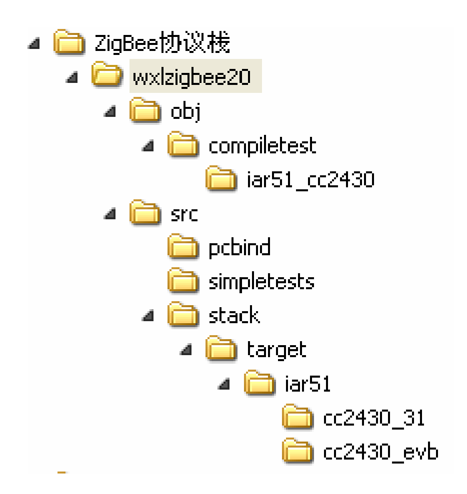
## 5.6.5、应用程序

```
void PingPong (void ) {
        apsFSM();//应用层无线数据收发处理
        switch (ppState) {
                case    PP_STATE_START_RX:    //  开始接收状态
                        ……………………………………………….
                        break;
                case PP_STATE_WAIT_FOR_RX:       //  等待接收
                        ………………………………………………..
                        Break;
                case PP_STATE_SEND:      //  开始发送状态
                        …………………………………………….
                        break;
                case PP_STATE_WAIT_FOR_TX:       //  等待发送
                        ………………………………………………
                        break;
        }
}
```

关于协议栈更多运行可以学习程序 apsFSM()。

# 5.7、实验步骤

## 5.7.1、熟悉协议栈路径



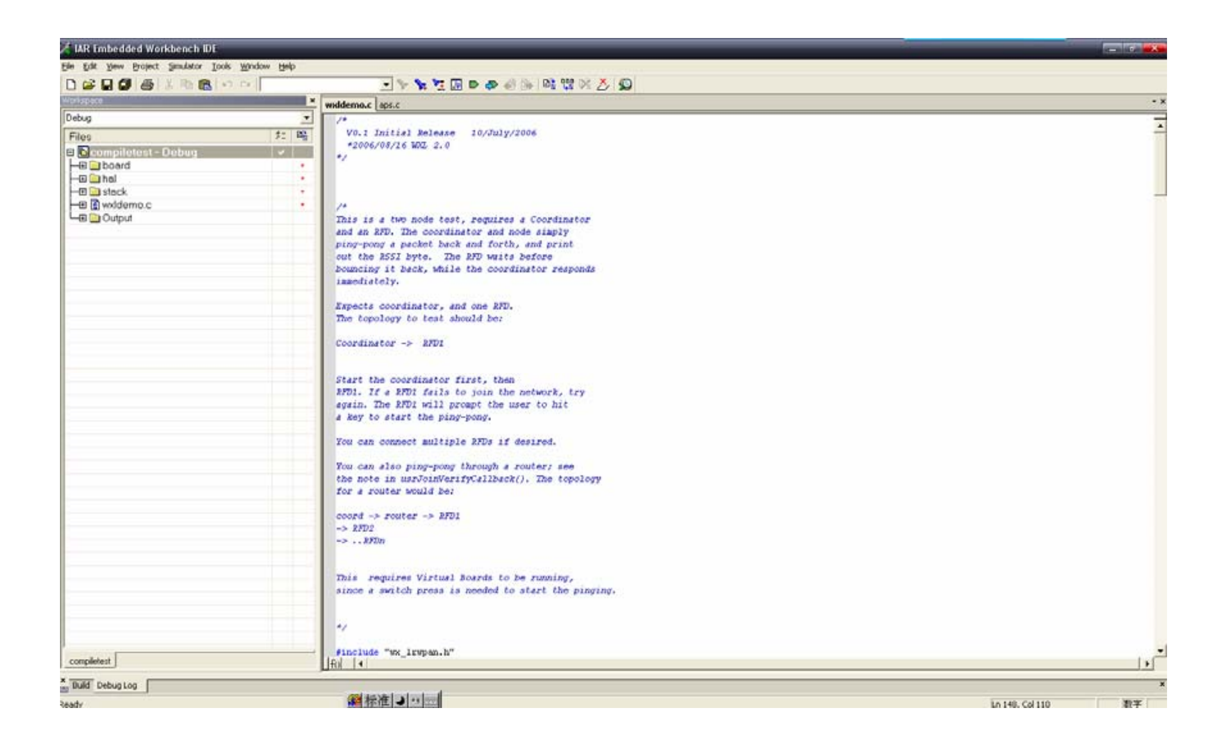

## 5.7.2、工程路径：

\ZigBee 协议栈\wxlzigbee20\obj\compiletest\iar51_cc2430

## 5.7.3、打开工程

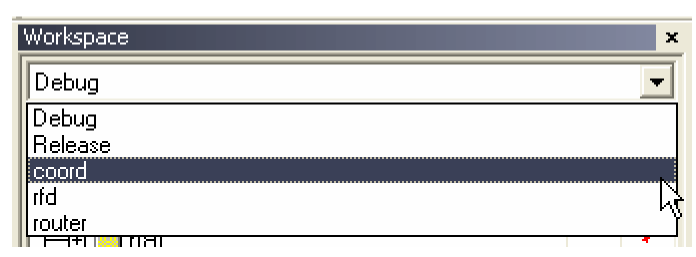


## 5.7.4、工程界面

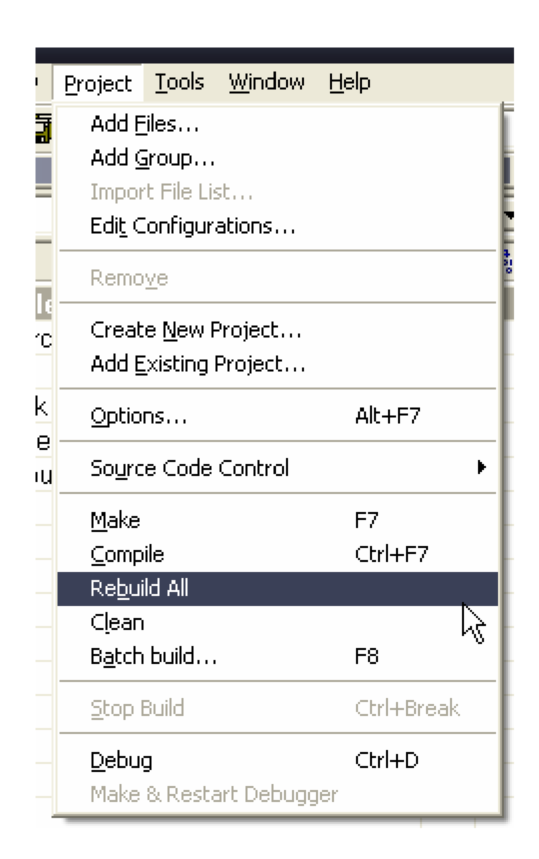



## 5.7.5、选择设备类型

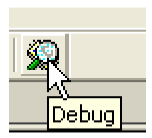
可以分别选择协调器（coord）、路由器（router）、终端（rfd）编译下载。



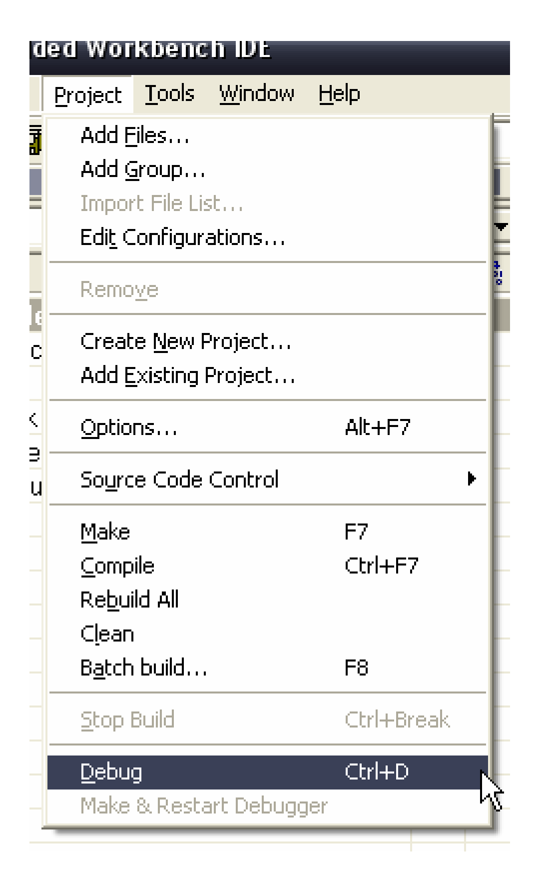
# 5.7.6、编译


Project
Tools
Window
Help
Add Files...
Add Group...
Import File List...
Edit Configurations...
Remove
Create New Project...
Add Existing Project...
Options... Alt+F7
Source Code Control
Make F7
Compile Ctrl+F7
Rebuild All
Clean
Batch build... F8
Stop Build Ctrl+Break
Debug Ctrl+D
Make & Restart Debugger

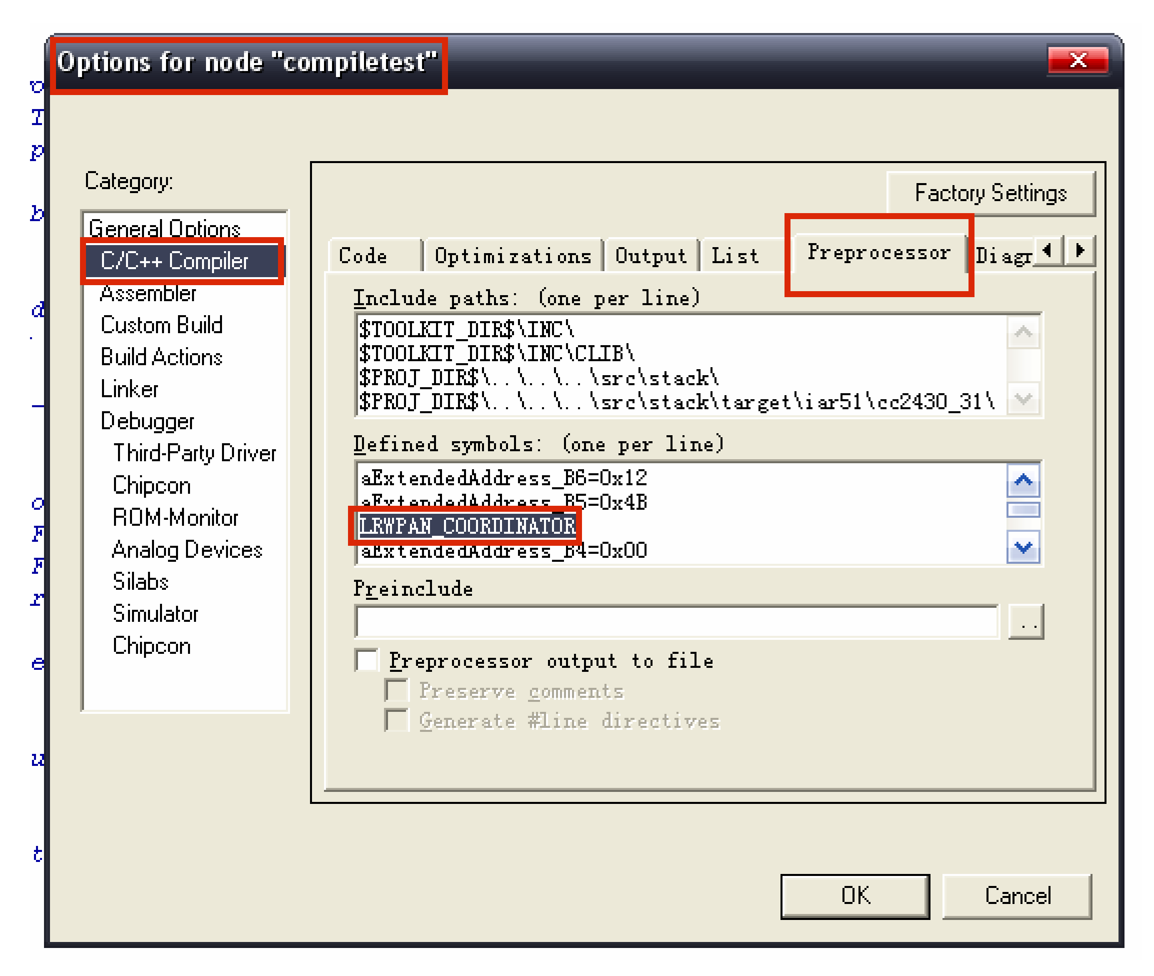
# 5.7.7、下载程序


Debug

或


ded Workbench IDE
Project
Tools
Window
Help
Add Files...
Add Group...
Import File List...
Edit Configurations...
Remove
Create New Project...
Add Existing Project...
Options... Alt+F7
Source Code Control
Make F7
Compile Ctrl+F7
Rebuild All
Clean
Batch build... F8
Stop Build Ctrl+Break
Debug Ctrl+D
Make & Restart Debugger

## 5.7.8、关于设备类型的定义在



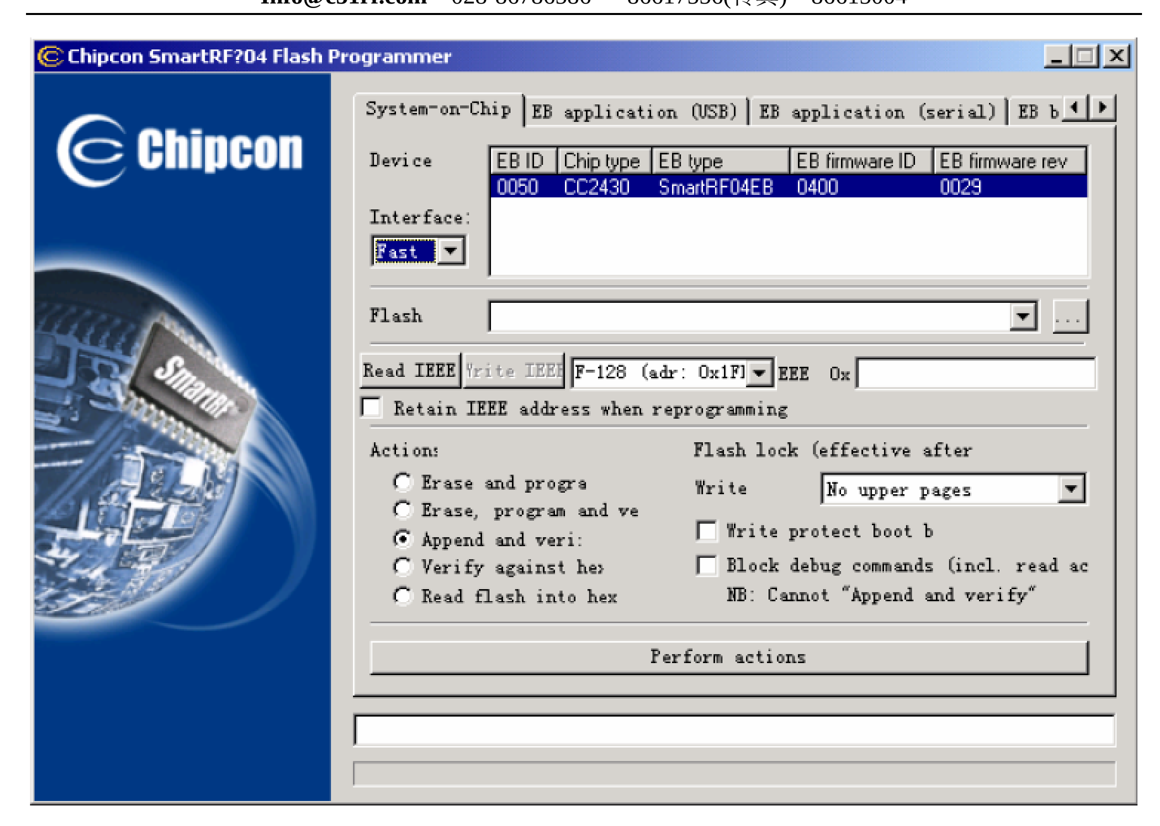

这里是条件编译定义。图中定义为协调器，路由器定义为：LRWPAN_ROUTER；没有定义就为终端设备类型。

## 5.7.9、64 位物理地址设定。



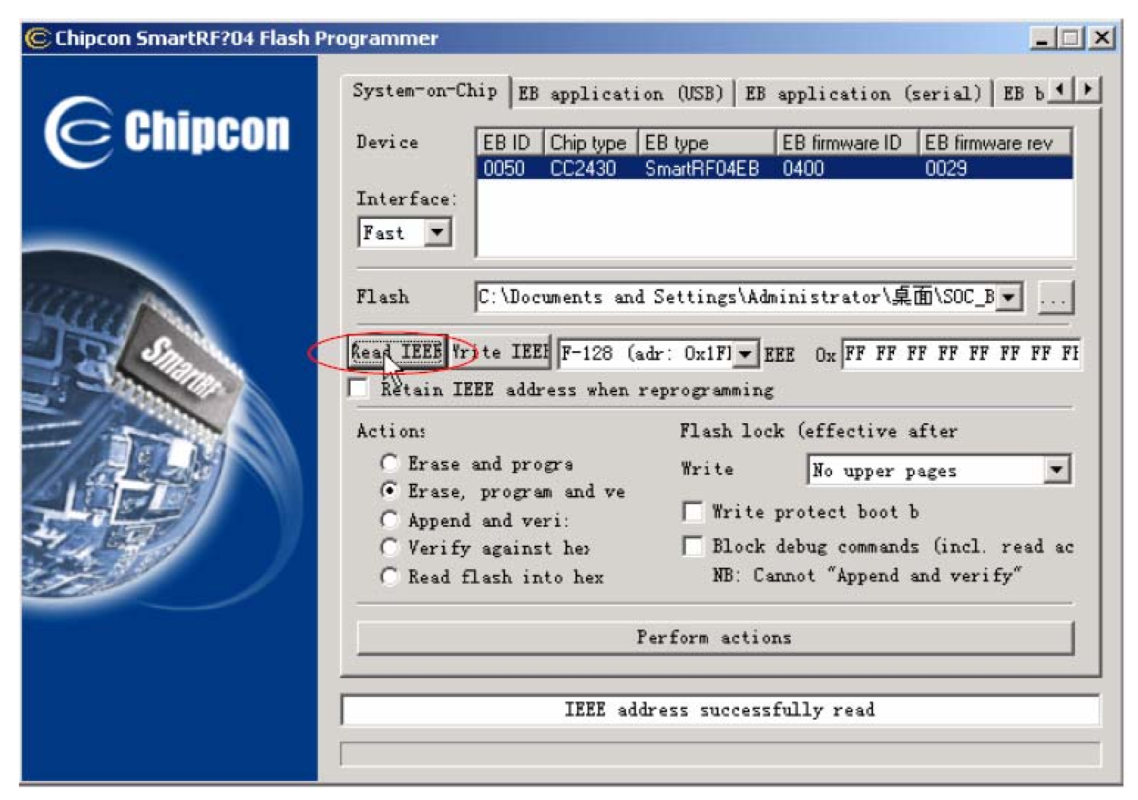

解压缩 ChipconFlashProgrammer_1_38.zip 文件可以看到 图标，双击这个图标打开 FLASH 烧写工具。

连接上 C51RF-3 型仿真器后再把模块接到仿真器可以看到如下显示：

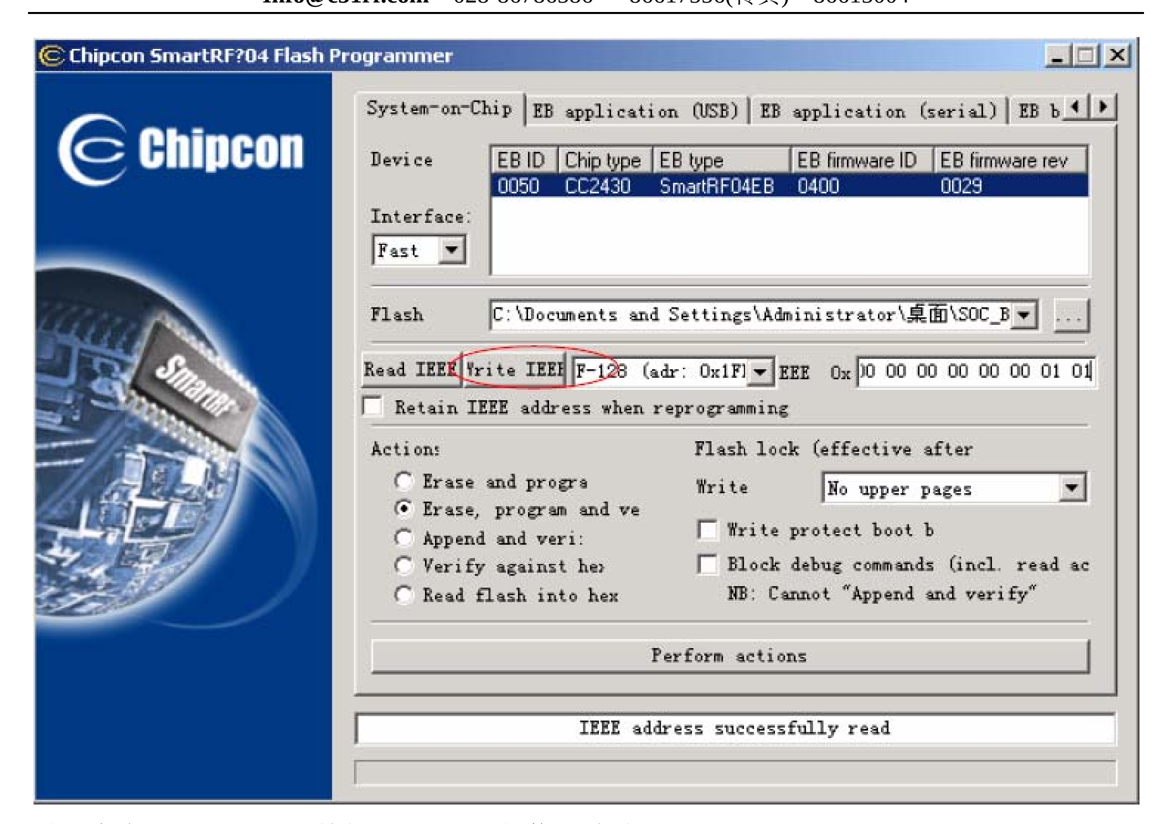



此时可以看到 FLASH 烧写工具已经检测到 CC2430 模块，如果此时没有检测到模块可以按仿真器的复位按键以便检测到模块。

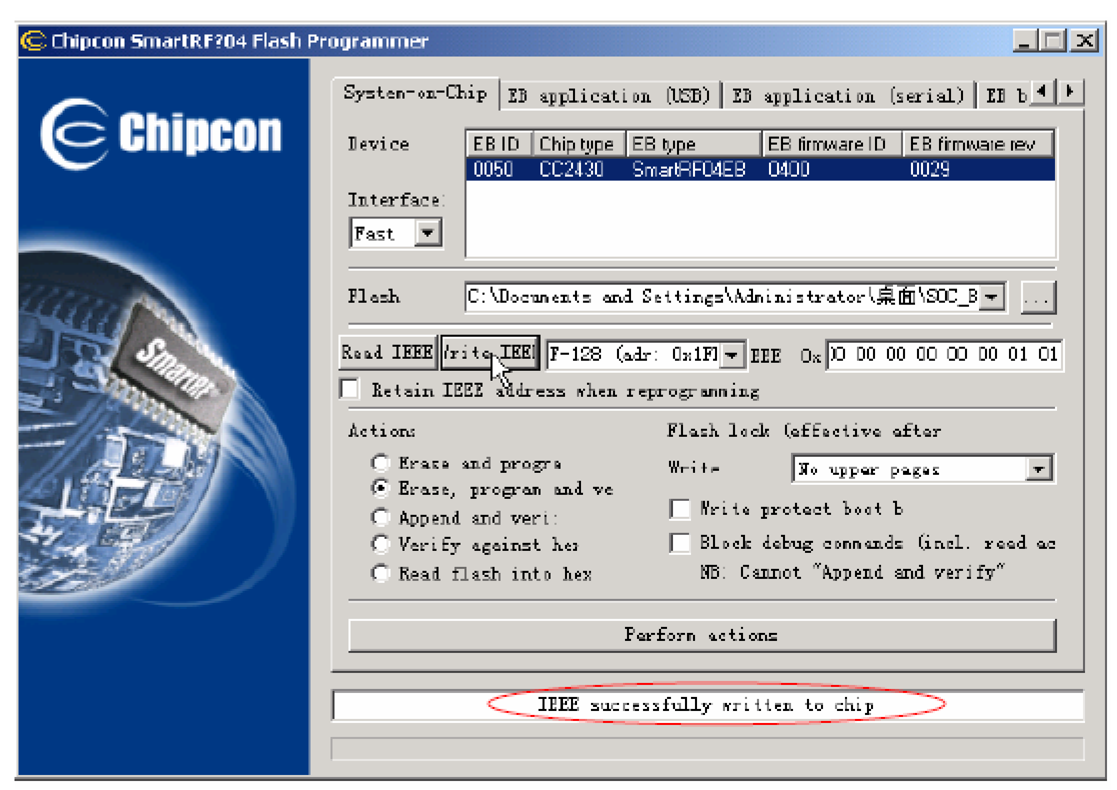
此时点击“Read IEEE”按键，可以读出模块的 64 位物理地址，如图：




然后把物理地址修改为你所要的地址：




最后点击“Write IEEE”按钮，写入64 位物理地址：




此时工具提示“IEEE successfully written to chip”表示地址写入成功。

## 5.8、实验结果

（A）：星型网络，

支持一个网络协调者和多个RFD 节点。

首先打开网络协调器模块的电源，此时模块的红色LED 点亮，表示模块建立网络成功。

然后打开RFD 模块的电源，此时观察网络协调器模块的红色LED 是否闪烁两次，如果有表示RFD 模块加入到网络中，此时RFD 的红色LED 点亮，并且绿色LED 跟网络协调者的模块开始同步闪烁。

（B）：串状网络。

首先打开网络协调器模块的电源，此时模块的红色LED 点亮，表示模块建立网络成功。

然后打开路由器模块的电源，此时观察网络协调器模块的红色LED 是否闪烁两次，如果有表示路由器模块加入到网络中，此时路由器模块的红色LED 点亮。

最后打开RFD 模块的电源，此时观察网络协调器模块的红色LED 和路由器的红色LED 是否闪烁两次。

如果路由器的红色LED 闪烁两次有表示RFD 模块加入到路由器中。

如果网络协调器模块的红色LED 闪烁两次有表示RFD 模块加入到网络协调器模块中。（此时应该把RFD 节点靠近路由器一些再打开电源，以便RFD 节点加入路由器节点组成串状网络）此时RFD 的红色LED 点亮，并且绿色LED 跟网络协调者的模块开始同步闪烁（RFD 节点通过路由器的网络协调器模块通信）。

如果现在关掉路由器节点的电源，RFD 节点和网络协调器的通信就会中断。

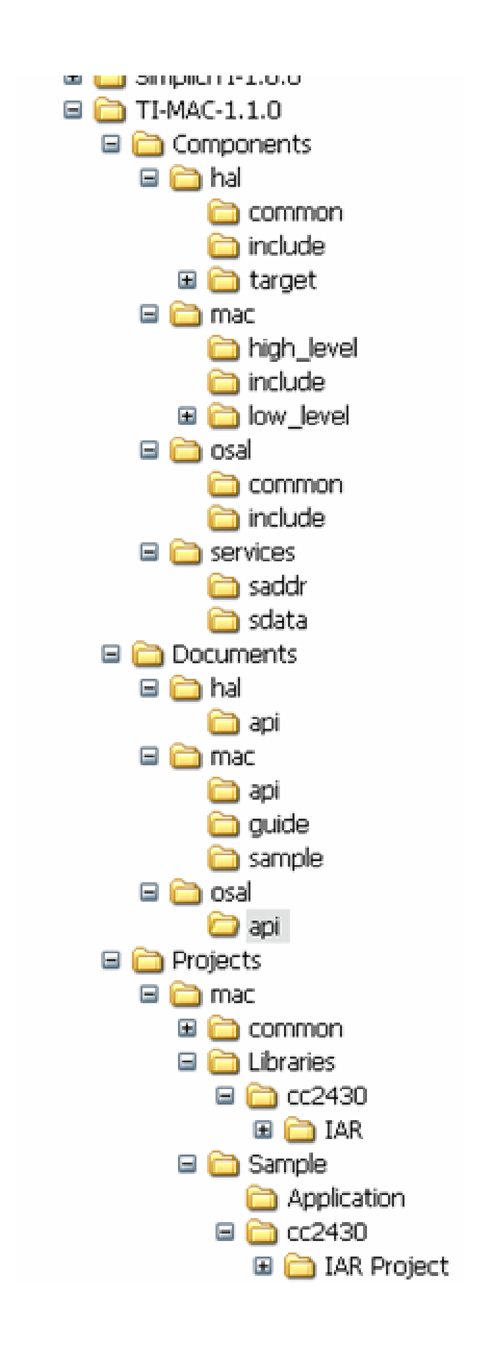
# 6、ZIGBEE2004 协议 UART0 中断

待补充：……………………………………………………

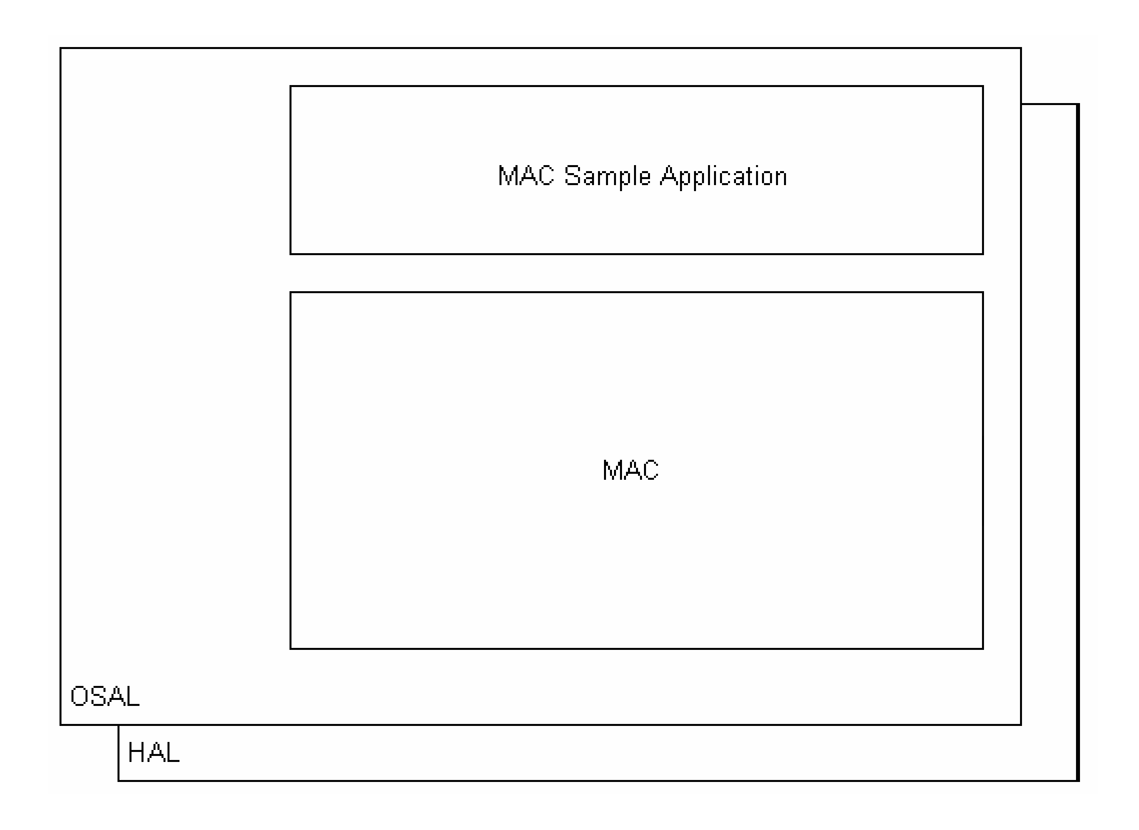
# 7、TI-MAC-1.1.0 使用说明手册

## 7.1、熟悉 MAC 例子

### 7.1.1、文件夹



## 7.1.2、MAC 层结构：




## 7.1.3、例子概述

主要是测试 802.15.4 的 MAC 层的操作，该例子通过判定按键事件完成下列任务：

- 设备运行和确定设备类型：协调器或设备
- 加入/格式化网络
- 连接/分离
- 扫描
- 发送和接收数据
- 信标支持

## 7.1.4、工程路径

C:\Texas Instruments\TI-MAC-1.1.0\Projects\mac\Sample\cc2430\IAR Project

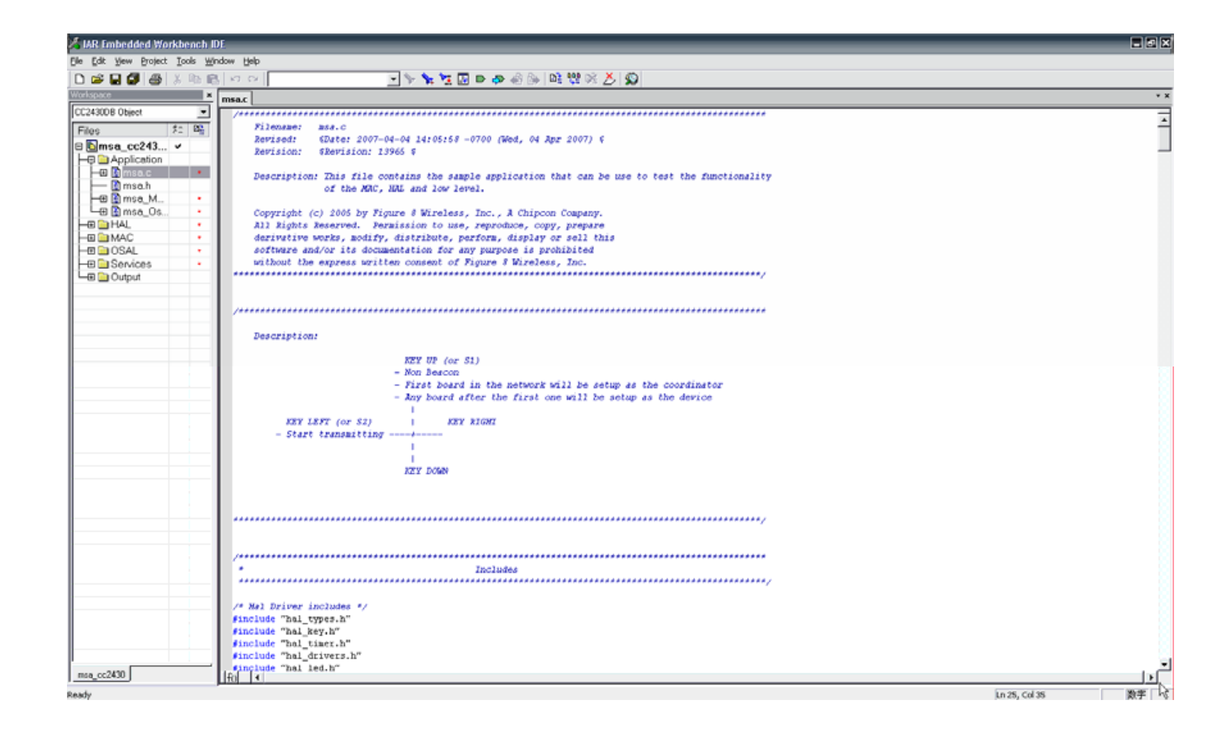




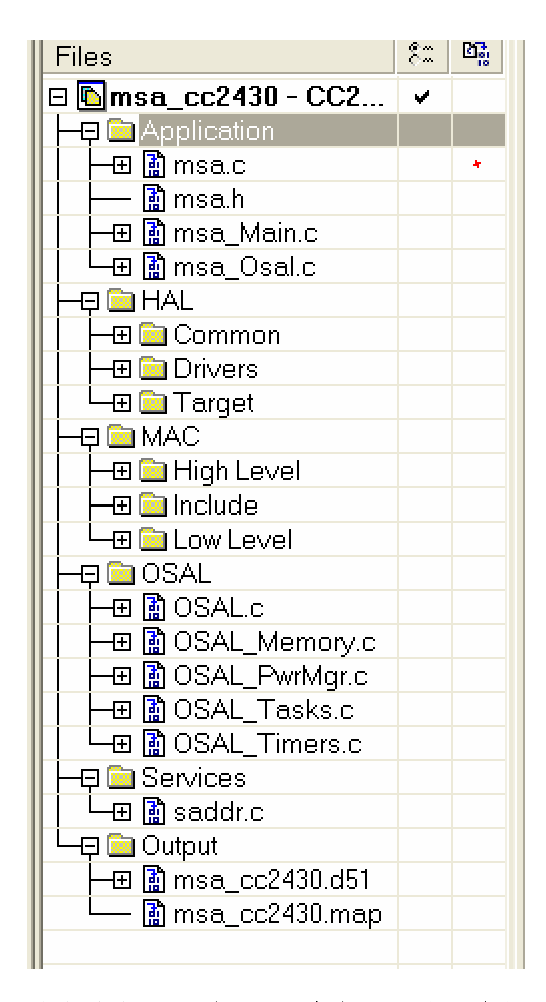




# 7.1.5、打开工程：


IAR Embedded Workbench IDE
File Edit View Project Tools Window Help

## 7.2、程序结构分析：




从上路上可以看出，程序主要分为 6 个部分：APP、HAL、MAC、OSAL、服务、输出。

### 7.2.1、HAL

HAL 是硬件抽象层。他主要是提供硬件接口服务，如 GPIO、定时器、UART 等。

### 7.2.2、OSAL

osal 是一个操作系统抽象层。它提供事件处理，消息传递，定时器，内存分配，和其他服务。

### 7.2..3、MAC

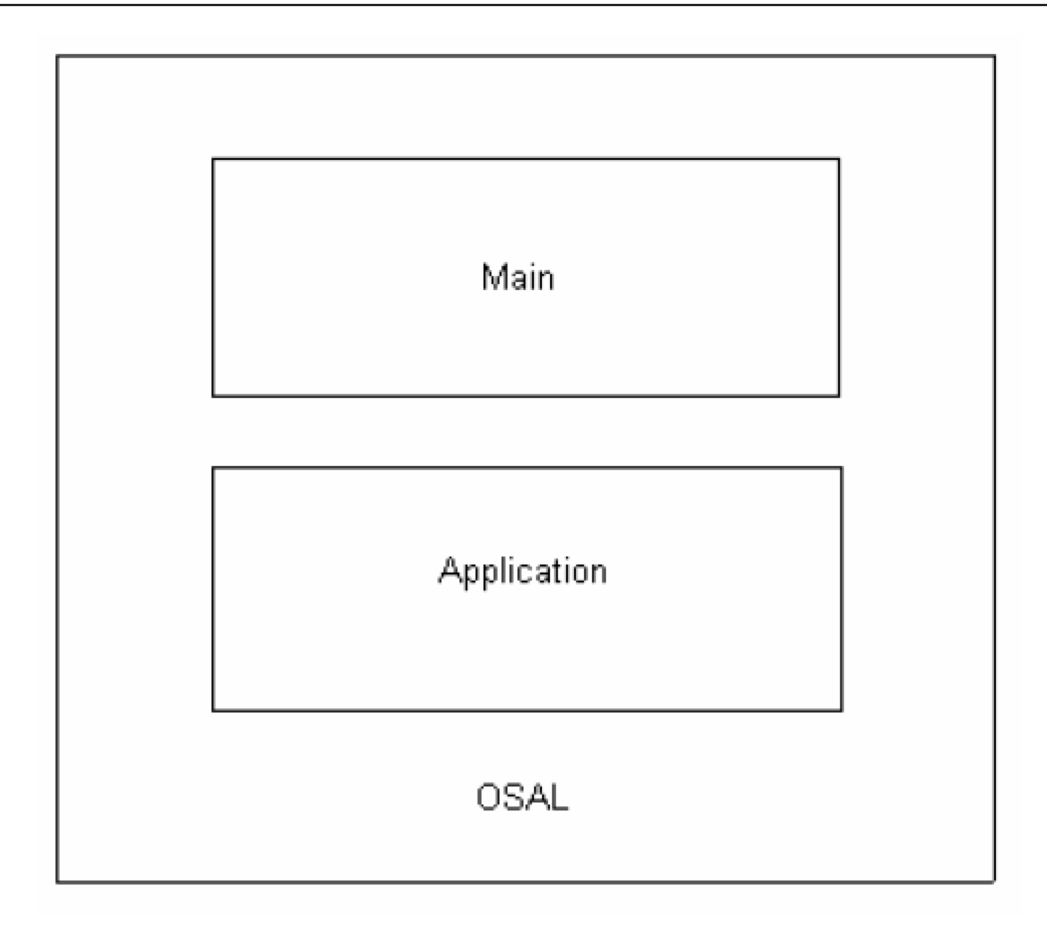
MAC 执行下列程序：

- 通过 API 函数处理 osal 事件和信息、接收命令帧、传输状态和定时器
- MAC 的功能为：扫描、连接和离开网络、网络开始、PAN ID 配置、间接数据、循环处理和信标处理；
- PIB 管理
- 建立并解析帧
- 射频管理
- MAC 定时器管理
- 帧的接收和传输，包括的 CSMA
- 帧确认处理
- 加密和解密
- 电源管理

## 7.3、程序分析

MAC 示例应用程序包含 4 个主要部分组成：主体，初始化，系统事件处理及反馈处理。

应用设计结构如下：




## 7.3.1、MAIN

主模块中包含的文件的初始化和启动系统。

**MSA_Main.c中的main ()**函数

整个程序运行的开始点，这时硬件、Mac 和 HAL 的初始化被执行。在这里 osal 初始化和添加任务。osal 计时器和 HAL 按键配置和启动。为了系统的正常工作，HAL 的驱动必须在此初始化。其他的初始化，如射频，中断和 Mac 也需要在这里进行。系统初始化完毕后，调用 osal_start_system （ ）开始运行任务。

## 7.3.2、应用分析

**MSA_Osal.c**文件
**osalAddTasks ()**
在这里添加任务，一共有三个任务：MAC、HAL 和应用任务被添加，并被初始化。用户可以指定每个的任务优先级，从 0-255 （最低-最高）。
**MSA_App.c**
**MSA_Init ()**

应用程序的初始化。 Mac 被初始化为设备或协调契。MAC 被复位，通过键盘事件来确定设备类型。task_id 指派一个价值用到 osaltaskadd （）函数 ;

**MSA_ProcessEvent ()**

应用信息处理。应用程序事件通过 osal 将处理两类事件：sys_event_msg 和所有其他不属于 sys_event_msg 。当 sys_event_msg 事件发生，收到的所有信息将逐一接收直到有最后一个信息被处理，然后返回到活动的任务管理。所有其他事件将接着一个个处理。

**MAC_CbackEvent ()**

这个回调函数发送的 Mac 时间到应用。应用程序必须执行这个功能。一个典型的执行功能是拨出 1 osal 信息，复制事件的参数信息，将消息发送到应用程序的 osal 事件处理程序。这个功能可能会从任务或中断的情况下被处理，因此，必须返回。

**MSA_CoordinatorStartup ()**

启动设备作为PAN协调器。作为协调器，MAC_RX_ON_WHEN_IDLE 为 TRUE，为了协调器可以接收信息。取决于对信标命令和超帧命令，协调器将启动信标允许或禁止。

**MSA_DeviceStartup ()**

启动设备作为普通设备。取决于对信标命令和超帧命令，协调器将启动信标允许或禁止。

**void MSA_AssociateReq ()**

调用 mac_mlmeassociatereq （ ）填充数据。

**MSA_AssociateRsp ()**

调用 mac_mlmeassociatereq （ ）填充数据。协调器将为每个设备分配短地址，设备接收这个短地址并做正确的配置设定。

**MSA_McpsDataReq ()**

调用 mac_mcpsdatareq （ ），输入参数，确定事件是直接或间接的，因此，正确的参数设置和数据将被发送。

**MSA_McpsPollReq ()**

调用 mac_mlmepollreq （ ）

**MSA_ScanReq ()**

调用l MAC_MlmeScanReq() 执行激活扫描

**MSA_BeaconPayLoadCheck()**

检查信标载荷，确定是否为 MSA 的类型。返回 TRUE 是，返回 FALSE 不是。

**MSA_DataCheck ()**

检查接收的数据是否与预先定义的数据匹配。

**MSA_HandleKeys ()**

处理键盘事件

## 7.3.3、操作

设置通道和传输速率

通道在msa.h文件中设置，如下参数：

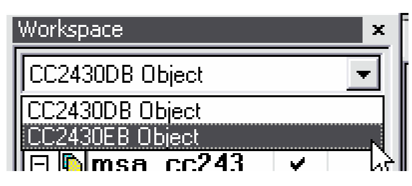
#define MSA_MAC_CHANNEL MAC_CHAN_11

| Name |
|---|
| MAC_CHAN_11 |
| MAC_CHAN_12 |
| MAC_CHAN_13 |
| MAC_CHAN_14 |
| MAC_CHAN_15 |
| MAC_CHAN_16 |
| MAC_CHAN_17 |
| MAC_CHAN_18 |
| MAC_CHAN_19 |
| MAC_CHAN_20 |
| MAC_CHAN_21 |
| MAC_CHAN_22 |
| MAC_CHAN_23 |
| MAC_CHAN_24 |
| MAC_CHAN_25 |
| MAC_CHAN_26 |
| MAC_CHAN_27 |
| MAC_CHAN_28 |

数据包发送间隔时间也在msa.h文件中设置，参数如下：（单位：毫秒）

```
#define MSA_WAIT_PERIOD 200
```

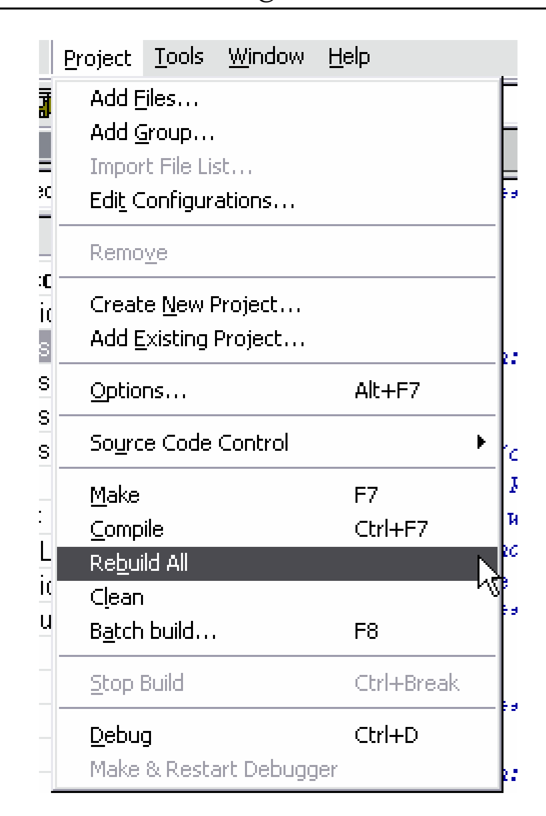
## 7.4、演示效果

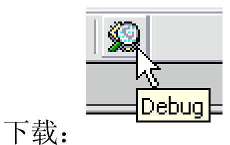






编译




下载：


第一个设备：通电，模块上的 G 灯快速闪烁，按下按键 S1（UP）、G 灯停止闪烁，此时格式化网络并建立网络。

第二个设备：通电，模块上的 G 灯快速闪烁，按下按键 S1（UP）、G 灯停止闪烁，一会又慢速闪烁，说明键入网络。

按下第二个设备的 S5（RIGHT），模块 R 灯快速闪烁、发送数据，第一个设备，模块 G 灯闪烁，回应收到数据。

再次按下第二个设备的 S5（RIGHT），停止发送数据，模块 G 灯回复慢速闪烁。第一个设备，模块 G 灯停止闪烁。

# 8、ZigBee2006 演示代码（见系统说明手册）

待补充：…………………………………………………………

# 9、Zigbee2006 串口互发

学习 Zigbee2006 发送、接收数据的方法，学习串口的使用，掌握 Zigbee 数据的处理方法。

## 9.1 实验目的

- 通过 Zigbee 实现计算机与计算机之间的通讯
- 学习 Zigbee 的数据处理方法
- 学习各自发送数据的方式

## 9.2 实验内容

通过串口发送数据，并通过串口接收数据，中间的介质采用 Zigbee2006.

## 9.3 实验设备

- C51RF-3-PK/C51RF-3-JKS(三个 CC2430 模块、开发底板两个、电源、USB 线)
- IAR 集成开发环境
- C51RF-3 仿真器

## 9.4 串口互发例程的实现

### 9.4.1 串口通信的实现

串口采用异步通讯方式，参数设计为 384000、8、n、1，具体设计方法请参考基础实验 13~16，这几个实验概括的串口的基本用法，本实验中，串口的接收部分延用了协议栈内容设计的接收处理函数函数，（SPIMgr.c 中的 void SPIMgr_ProcessZToolData ( uint8 port, uint8 event )函数），而发送函数在本实验中代码如下。

```
//cd wxl          串口 0 发数据
#include <ioCC2430.h>
#include <wxl_uart.h>
//#include <string.h>

/****************************************************************
*函数功能 ：初始化串口 1
*入口参数 ：无
*返 回 值 ：无
*说      明 ：57600-8-n-1
```

```
****************************************************************/
void initUARTtest(void)
{

    CLKCON &= ~0x40;                    //晶振
    while(!(SLEEP & 0x40));       //等待晶振稳定
    CLKCON &= ~0x47;                    //TICHSPD128 分频，CLKSPD 不分频
    SLEEP |= 0x04;        //关闭不用的 RC 振荡器
    PERCFG = 0x00;                      //位置 1 P0 口
    P0SEL |= 0x0C;                  //P0 用作串口
    P2DIR &= ~0xC0;                               //P0 优先作为串口 0
    U0CSR |= 0x80;                  //UART 方式
    UTX0IF = 0;

}
/****************************************************************
*函数功能 ：串口发送字符串函数
*入口参数 : data:数据
*             len :数据长度
*返 回 值 ：无
*说    明 ：
****************************************************************/
void UartTX_Send_String(char *Data,int len)
{
  int j;
  for(j=0;j<len;j++)
  {
    U0DBUF = *Data++;
    while(UTX0IF == 0);
    UTX0IF = 0;
  }
}
void UartTX_Send_Single(char single_Data)
{
    U0DBUF = single_Data;
    while(UTX0IF == 0);
    UTX0IF = 0;
}
/****************************************************************************
描述：
    串口接收一个字符
函数名：char UartRX_Receive_Char (void)
****************************************************************************/
char UartRX_Receive_Char (void)
```

```
{
    char c;
    unsigned char status;
    status = U0CSR;
    U0CSR |= UART_ENABLE_RECEIVE;
    while (!URX0IF);
    c = U0DBUF;
    URX0IF = 0;
    U0CSR = status;
    return c;
}
/*******************************************************************************
描述：
    波特率的设置
函数名：void Uart_Baud_rate(int Baud_rate)
*******************************************************************************/
void Uart_Baud_rate(int Baud_rate)
{
  switch (Baud_rate)
  {
    case 24:
      U0GCR |= 6;
      U0BAUD |= 59;                 //波特率设置
    break;
    case 48:
      U0GCR |= 7;
      U0BAUD |= 59;                 //波特率设置
    break;
    case 96:
      U0GCR |= 8;
      U0BAUD |= 59;                 //波特率设置
    break;
    case 144:
      U0GCR |= 8;
      U0BAUD |= 216;                //波特率设置
    break;
    case 192:
      U0GCR |= 9;
      U0BAUD |= 59;                 //波特率设置
    break;
    case 288:
      U0GCR |= 9;
      U0BAUD |= 216;                //波特率设置
    break;
```

```
        case 384:
          U0GCR |= 10;
          U0BAUD |= 59;                    //波特率设置
        break;
        case 576:
          U0GCR |= 10;
          U0BAUD |= 216;                   //波特率设置
        break;
        case 768:
          U0GCR |= 11;
          U0BAUD |= 59;                    //波特率设置
        break;
        case 1152:
          U0GCR |= 11;
          U0BAUD |= 216;                   //波特率设置
        break;
        case 2304:
          U0GCR |= 12;
          U0BAUD |= 216;                   //波特率设置
        break;
        default:
        break;
    }
}
```

## 9.4.2 发送函数

发送数据采用短地址的方式发送，在该函数中设计的传递参数有发送的数据、目的地址、数据长度。实现的代码如下：

```
//*********************************************************************
//**以短地址方式发送数据
//buf ::发送的数据
//addr::目的地址
//Leng::数据长度
//*********************************************************************
uint8 SendData(uint8 *buf, uint16 addr, uint8 Leng)
{
        afAddrType_t SendDataAddr;
        SendDataAddr.addrMode = (afAddrMode_t)Addr16Bit;            //短地址发送
        SendDataAddr.endPoint = SAMPLEAPP_ENDPOINT;
        SendDataAddr.addr.shortAddr = addr;
            if ( AF_DataRequest( &SendDataAddr, //发送的地址和模式
                               &SampleApp_epDesc,     //终端（比如操作系统中任务 ID 等）
                               2,//发送串 ID
```

```
                                Leng,
                                buf,
                                &SampleApp_TransID,   //信息 ID（操作系统参数）
                                AF_DISCV_ROUTE,
                              //  AF_ACK_REQUEST,
                                AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
      {
          return 1;
      }
      else
      {
          return 0;// Error occurred in request to send.
      }
  }
```

## 9.4.3 接收处理函数

接收处理函数是将接收的数据进行处理，在实验中不同的设备收到的数据由不同的处理方式。

协调器收到数据后，首先判断是不是由新节点加入的网络，如果有的话，判断设备的类型，知道类型后，判断是不是已经加入了网络，如果是，就不再处理，如果不是，将该节点的物理地址和网络地址放到一个 buffer 里面。小灯闪烁两次。

如果不是新节点加入，将数据直接通过串口发送出去。

如果是路由器收到数据后，直接将数据通过串口发送。本实验中没有对终端节点处理。

```
void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  int new_node_flag;
  new_node_flag = 0;
  if(clear == 0)
  {
    JoinNode.RouterCount = 0;                  //路由器计数清零
    JoinNode.RfdCount = 0;                      //终端计数清零
    clear = 1;
  }
  switch ( pkt->clusterId )
  {
    case SAMPLEAPP_PERIODIC_CLUSTERID:
      break;

    case SAMPLEAPP_FLASH_CLUSTERID:
      memcpy(RxBuf,pkt->cmd.Data,pkt->cmd.DataLength);
#if defined( ZDO_COORDINATOR )             //如果是协调器收到数据
      if((RxBuf[0] == 'n') && (RxBuf[1] == 'e') &&(RxBuf[2] == 'w'))        //新节点加入
      {
```

```
            if((RxBuf[3] == 'R') && (RxBuf[4] == 'O') && (RxBuf[5] == 'U'))    //判断是路由器节点
            {
              for(int i=0;i<JoinNode.RouterCount;i++)
              {
                for(int j=0;j<8;j++)
           {
                  //判断是否有相同地址
                  if(JoinNode.RouterAddr[JoinNode.RouterCount][j] == RxBuf[j+6])
                  {
                    new_node_flag++;              //判断位相同标志加 1
                  }
                  else
                  {
                    new_node_flag = 0;            //判断位不同，表示地址不同，标志清 0
                    j += 8;
                  }
                }
                if(new_node_flag == 8)
                {
                  i += JoinNode.RouterCount;                       //退出查询
                }
              }
              if(new_node_flag == 0)
              {
                for(int i=0;i<8;i++)
                {
                  JoinNode.RouterAddr[JoinNode.RouterCount][7-i] = RxBuf[i+6];       //存放物理地址
                }
                JoinNode.RouterAddr[JoinNode.RouterCount][8] = RxBuf[6+8];          //存放网络地址
                JoinNode.RouterAddr[JoinNode.RouterCount][9] = RxBuf[6+9];
                JoinNode.RouterCount ++;
              }
            }
            UartTX_Send_String( RxBuf,6);
             //通过串口发送数据
            UartTX_Send_String( JoinNode.RouterAddr[JoinNode.RouterCount-1],10);
          }
          else
          UartTX_Send_String(RxBuf,pkt->cmd.DataLength);
#elif defined( RTR_NWK ) && (!defined(ZDO_COORDINATOR))                //选择路由器
          UartTX_Send_String(RxBuf,pkt->cmd.DataLength);                  //通过串口发送数据
#else                                                          //剩下的就是终端节点
#endif
          HalLedBlink( HAL_LED_4, 2, 50, 100 );                        //小灯闪烁
```

```
            break;
        }
    }
```

## 9.4.4 串口接收处理函数

串口接收到数据后，如果是协调器设备则判断有没有相同地址（在发送的数据的时候需要在数据前加上接收设备的物理地址），如果由则发送此数据（去除物理地址），如果不是则不处理。

如果是路由器则直接将数据发送到网管（在一个网络中，网络的网络地址为 0x0000）。

```
void SPIMgr_ProcessZToolData ( uint8 port, uint8 event )
{
    int s;
    Uart_len = 0;
#ifdef ZDO_COORDINATOR
    int k,f;
    int new_node_flag = 0;
#endif

    /* Verify events */
    if (event == HAL_UART_TX_FULL)
    {
        // Do something when TX if full
        return;
    }

    if (event & (HAL_UART_RX_FULL | HAL_UART_RX_ABOUT_FULL |
HAL_UART_RX_TIMEOUT))
    {
        while (Hal_UART_RxBufLen(SPI_MGR_DEFAULT_PORT))
        {
            HalUARTRead (SPI_MGR_DEFAULT_PORT, &Uart_Rx_Data[Uart_len], 1);     //读取串口数据
        switch (state)
            {
                case SOP_STATE:
                    if (Uart_Rx_Data[Uart_len] == SOP_VALUE)
                        state = CMD_STATE1;
                    break;
                case CMD_STATE1:
                    CMD_Token[0] = Uart_Rx_Data[Uart_len];
                    state = CMD_STATE2;
                    break;
                case CMD_STATE2:
                    CMD_Token[1] = Uart_Rx_Data[Uart_len];
```

```
                state = LEN_STATE;
                break;
            case LEN_STATE:
                LEN_Token = Uart_Rx_Data[Uart_len];
                if (Uart_Rx_Data[Uart_len] == 0)
                    state = FCS_STATE;
                else
                    state = DATA_STATE;
                tempDataLen = 0;
                // Allocate memory for the data
                SPI_Msg = (mtOSALSerialData_t *)osal_msg_allocate( sizeof ( mtOSALSerialData_t ) +
2+1+LEN_Token );
                if (SPI_Msg)
                {
                    // Fill up what we can
                    SPI_Msg->hdr.event = CMD_SERIAL_MSG;
                    SPI_Msg->msg = (uint8*)(SPI_Msg+1);
                    SPI_Msg->msg[0] = CMD_Token[0];
                    SPI_Msg->msg[1] = CMD_Token[1];
                    SPI_Msg->msg[2] = LEN_Token;
                }
                else
                {
                    state = SOP_STATE;
                    return;
                }
                break;
            case DATA_STATE:
                    SPI_Msg->msg[3 + tempDataLen++] = Uart_Rx_Data[Uart_len];
                    if ( tempDataLen == LEN_Token )
                        state = FCS_STATE;
                break;
            case FCS_STATE:
                FSC_Token = Uart_Rx_Data[Uart_len];
                //Make sure it's correct
                if ((SPIMgr_CalcFCS ((uint8*)&SPI_Msg->msg[0], 2 + 1 + LEN_Token) == FSC_Token))
                {
                    osal_msg_send( MT_TaskID, (byte *)SPI_Msg );
                }
                else
                {
                    // deallocate the msg
                    osal_msg_deallocate ( (uint8 *)SPI_Msg);
                }
```

```
                //Reset the state, send or discard the buffers at this point
                state = SOP_STATE;
                break;
              default:
                break;
            }
            Uart_len++;
        }
#ifdef ZDO_COORDINATOR
                for(k=0;k<JoinNode.RouterCount;k++)
                {
                  for( s=0;s<8;s++)
                  {
                    if(JoinNode.RouterAddr[k][s] == Uart_Rx_Data[s])          //判断是否有相同地址
                    {
                      new_node_flag++;          //判断位相同标志加 1
                    }
                    else
                    {
                      new_node_flag = 0;                     //判断位不同，表示地址不同，标志清 0
                      s += 8;
                    }
                  }
                    if(new_node_flag == 8)
                    {
                      f = k;
                      Short_Add = JoinNode.RouterAddr[k][9];                    //取短地址地位
                      k += JoinNode.RouterCount;
                      Short_Add <<= 8;                                          //退出查询
                    }
                }
                if(new_node_flag == 8)
                {
                  Short_Add |= JoinNode.RouterAddr[f][8];                       //取短地址高位
                  for(s=0;s<(Uart_len - 8);s++)
                  {
                    RfTx.TXDATA.DataBuf[s] = Uart_Rx_Data[8+s];
                     //取数据前 8 位是物理地址这里用 ASCII 表示
                  }
                  SendData(RfTx.TXDATA.DataBuf,Short_Add,Uart_len-8);             //发送数据
                }
#elif defined( RTR_NWK ) && (!defined(ZDO_COORDINATOR))
                  for(s=0;s<Uart_len;s++)
                  {
```

```
                RfTx.TXDATA.DataBuf[s] = Uart_Rx_Data[s];        //取串口接收的数据到发送 buf 中
              }
              SendData(RfTx.TXDATA.DataBuf,0x0000,Uart_len);     //将所有数据到网管
#else
#endif
   }
}
#endif //ZTOOL
```

## 9.6 代码实现

略。（请参考具体源代码，内部有详细的注释，如有疑问请联系我公司技术支持，028-86786586-157/158）

本例程对无线串口通信进行了诠释，在稍加修改后，就可以实现点对点，点对多点，单播、广播等不同的发送方式，本手册只为一个使用向导，具体功能实现请参考源代码，在源代码中有纤细的注释。

## 9.7 实验步骤

6. 连接好硬件，请参阅系统说明书。
7. 根据使用说明书的方法下载协调器代码，协调器代码运行后黄灯常亮，红灯闪烁 4 次后熄灭，表示网络已经建立成功，并通过串口发送”haha!Nework found succeed”。此时协调器处于等待节点加入的状态。
8. 用同样的方法下载路由器代码，路由器运行加入网络后，黄灯常亮，红灯闪烁两次后熄灭，路由器会通过串口发送“haha!Rou jiond succeed”，同时协调器在收到加入信息后红灯也闪烁两次，并将路由器的地址信息发送到 PC“newROU00000005┌”（”new”:表示新节点、“ROU”表示是路由器，“00000005”：表示设备的物理地址。在数据中的“┌”是非可见字符，他是网络地址在串口助手中用 16 进制的方式可以看到它的数据为 0x0001）。
9. 如果协调器向路由器发送数据，数据前必须要加入路由器设备的物理地址（这里是通过物理地址去寻到网络地址，再通过网络地址发送，这是因为在网络中协调器没有存储信息，如果设备调电，网络地址可能改变），以上面的新节点数据为例，如果发送的数据为“hello world！”哪么发送的数据为“00000005hello world！”。
10. 如果路由器向协调器发送数据，则直接发送即可，因为协调器的网络地址不会改变，为“0x0000”所以他们是一个透明传输过程。

# 10、ZigBee 2006 绑定实验

-------简单 ZigBee 2006 应用接口（simple api）

## 10.1、实验目的

- 设置这些设备自动的进入网络
- 创建从每一个开关到一个或多个灯的绑定
- 从开关设备发送一个改变灯状态的命令
- 为某个开关到不同的灯从新指派绑定
- 之后增加新的灯或开关到该网络

## 10.2、实验原理

关于详细的程序清单见 SAPI.C 文件。

- 初始化
  - ZB 系统复位（zb_SystemReset）
  - ZB 启动请求（zb_StartRequest）
- 配 置
  - ZB 读配置（zb_ReadConfiguration）
  - ZB 写配置（zb_ WriteConfiguration）
  - ZB 获得设备信息（zb_GetDeviceInfo）
- 发现（设备，网络和服务发现）
  - ZB 发现设备请求（zb_FindDeviceRequest）
  - ZB 绑定设备请求（zb_BindDeviceRequest）
  - ZB 允许绑定请求（zb_AllowBindRequert）
  - ZB 许可加入请求（zb_PermitJoinRequest）
- 数据传输
  - ZB 发送数据请求（zb_SendDataRequest）
  - ZB 接收数据指示（zb_ReceiveDataIndication）

### 10.2.1、网络形成

协调器将扫描所有被 ZCD_NV_CHANLIST 参数指定的通道和选择一个最少能量的通道。如果有两个及以上的最小能量通道，协调器选择在 ZB 网络中存在的序号最小的通道。协调器将选择用 ZCD_NV_PANID 参数指定的网络 ID。路由器和终端设备将扫描用 ZCD_NV_CHANLIST 配置参数制定的通道和试图发现 ID 为 ZCD_NV_PANID 参数指定的网络。

**协调器格式化网络**

可以调用下面函数来格式化网络：

NLME_NetworkFormationRequest(

zgConfigPANID,

```
                    zgDefaultChannelList,
                    zgDefaultStartingScanDuration,
                    beaconOrder,
                    superframeOrder,
                    false
                    );
```

一般不直接用上面的函数形成网络，而是用ZDO_StartDevice()函数来启动一个设备。

**路由器和终端设备加入网络**

发现一个网络将调用下面函数：

```
NLME_NetworkDiscoveryRequest(
                    uint32 ScanChannels,
                    byte scanDuration
                    );
```

发现网络存在后，就调用下面函数加入该网络：

```
NLME_OrphanJoinRequest(
                uint32 ScanChannels,
                byte ScanDuration
                );
```

最好也不要用这两个函数去发现/加入网络，而是用ZDO_StartDevice()启动该设备。

**ZDO_StartDevice**

功能描述：在网络中启动设备，协调器、路由器、终端设备都可以用该函数启动，启动之后，设备根据自身的类型去建立或发现和加入网络。

函数原型：

```
ZDO_StartDevice(
                byte logicalType,
                devStartModes_t startMode,
                byte beaconOrder,
                byte superframeOrder
              );
```

## 10.2.2、绑定

绑定是控制信息从一个应用层到另一个应用层流动的一种机制。在 ZB06 版本中，绑定机制在所有的设备中被执行。

**绑定建立**

这里可以直接调用函数 zb_BindDeviceRequest()发起绑定请求：

```
zb_BindDevice (
            uint8 create,          //是否创建绑定，TRUE 创建，FALSH 解除
            uint16 commandId,  //命令 ID，基于某命令的绑定
            uint8 *pDestination  //指向扩展地址指针
```

);

函数 APSME_BindRequest 函数创建绑定。

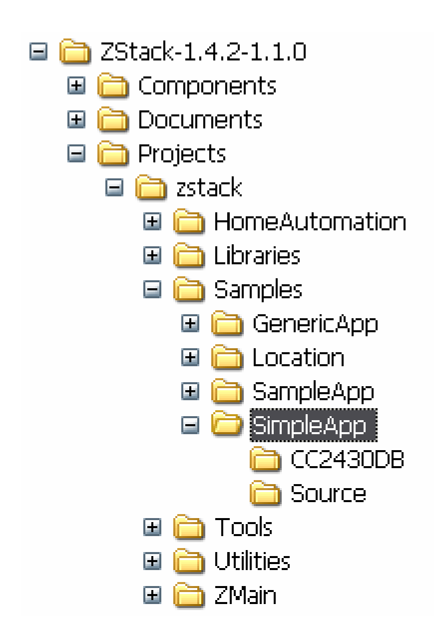
绑定解除

解除绑定和建立绑定请求函数都是 zb_ BindDeviceRequest()，但是参数不一样（第一个参数为 FALSH），

### 10.2.3、命令

命令就是为了实现某中特定的通信，而指定的一种强制性的通信方式。

## 10.3、例子路径




工程路径：
C:\Texas Instruments\ZStack-1.4.2-1.1.0\Projects\zstack\Samples\SimpleApp\CC2430DB

## 10.4、灯开关实验

### 10.4.1、试验介绍

**设备（Devices）**

该示范例子有两种应用设备类型----开关和灯。

应用例子工程有作为终端设备（end-device）的简单开关配置和作为协调器或路由器设备的简单管理器配置。

当这个设备第一次开启的时候，它进入一个“保持状态”，LEDx 闪烁。

对于灯管理器设备，在该状态下，按下 SW1 它将使该设备作为协调器启动，期间要是按下 SW2 她将使该设备作为路由器启动。

对于开关设备而言，在该状态下，无论是按下 SW1 还是 SW2 都将作为终端设备启动。

**命令：**有一个单一的应用命令----一个“拨动”（TOGGLE）命令。对于开关该命令作为输出被定义，对于管理器却作为输入别定义。该命令信息除了命令标志符之外没有其他参数。

**绑定：**“按钮”绑定被使用。

在一个开关和一个管理器间绑定被创建，首先是这个管理器要进入允许绑定模式。接着是开关（在一定时间内）发出一个绑定请求。这就将从开关到管理器之间创建一个绑定。

重复上面的过程，一个开关可以与多个管理器绑定。

为某个开关重新分配绑定，这个绑定请求与同一个删除参数被发出。这就将该开关的所有绑定移除。现在就可以用上面的绑定方法重新与其他的管理器进行绑定操作。

针对简单管理器和简单开关的配置编程有详细的描述。确保只能有一个管理器作为协调器，其他都作为路由器。

设备自动加入网络之后（LED3 闪亮---开关设备，协调器建立网络 LED3 点亮，路由器加入网络 LED3 点亮），采用下面的控制方式来创建绑定：

- 通过按某个管理器的 S1 使它进入允许绑定模式。
- 在某个灯开关上按下 S1（10 秒之内）发出绑定请求。
- 这就将使该开关设备绑定到该（处于绑定模式下的）管理器设备上。
- 当开关绑定成功时，（开关设备上的）LED1 闪亮。
- 之后，开关设备上的 S4 被按下就将发送“切换”命令。它将使对应的管理器设备上的 LED1 状态切换。
- 如果开关设备上的 S3 按下，它将移除该设备上所有的绑定。

## 10.4.2、实验步骤及结果

1、 找到工程目录打开：

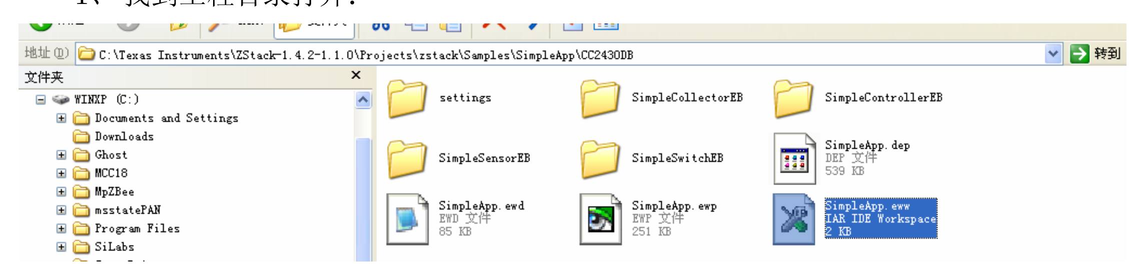



图 7.1 simple 工程路径

2、 编译下载：

用两个模块，选择管理器编译下载：

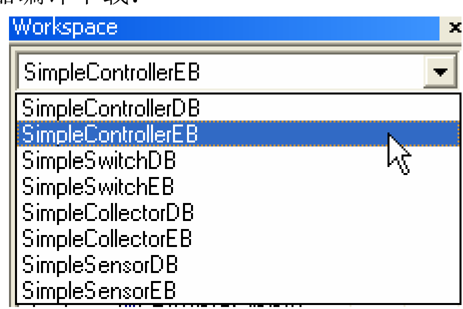



图 7.2 选择管理器（灯）

换个模块，选择开关设备编译下载：

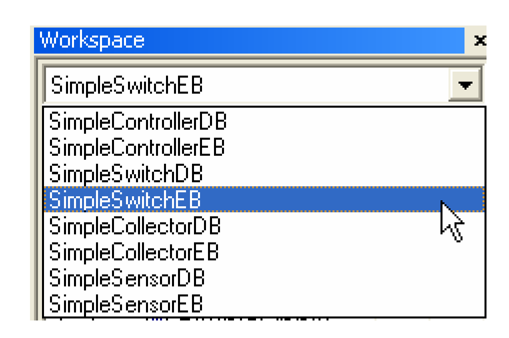



图 7.2 选择开关

3、 选择适当的类型启动，这里三个模块，管理器两个分别作为协调器和路由器启动，开关设备作为终端设备启动。

4、 建立绑定。可以在协调器和终端设备之间建立绑定，或可以在路由器和终端设备间建立绑定，开关也可以同时绑定所有的管理器设备，其绑定方法如前介绍。

5、 绑定之后，就可以在建立绑定之间的设备发送命令，可以观察管理设备灯 LED1 的显示状态的变化。

6、 按下 S2 可以解除开关上的所有绑定，从而可以按照 4~5 步从新绑定和传输命令。

## 10.4.3、实验总结

下面介绍用简单的 API 开发一个应用的方法：

● 确定应用中的所有的设备

– 如：温度传感器，占有传感器，自动调温器，加热单元和遥控

– 为他们中的每一个设备都分配一个设备ID（唯一的16位ID）

●确定“命令”，为每个设备在设备之间交换信息分配的16位命令ID，例如：

– 读温度

– 读自动调温器设置

– 加热/制冷单元的控制

● 为每个“命令”，确定设备“引起”（输出）和“吸引”（输入）命令。

– 温度读是“输出”从占有传感器设备和“输入”到自动调温器

– 等等

● 为每个设备创建简单描述符结构。这个包括：

– 为每个设备分配设备ID和版本。

– 为设备指定“输出”和“输入” 命令列表

– 指定一个模式（profile）ID。这个是唯一的应用模式16位的身份识别码，由ZB联盟分配。

●为每一个“命令”

– 定义信息交换格式和它的解释

– 例如：温度值能被交换作为

( 格式 ) “*一个8位值*”

( 解释 ) “0 指示0°C 和 255指示 64°C 步长为 0.25°C ”

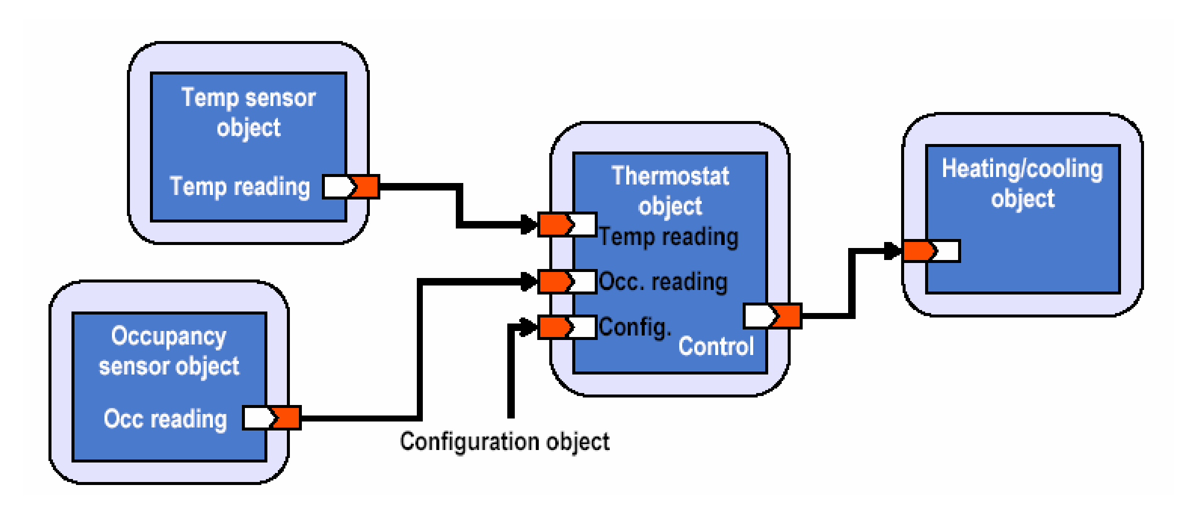



图 7.4 项目配置示意图

- 为每个设备写应用程序

– “输出”命令设备应该能产生一个数据包，作为周期发送或外部时间触发发送。

– “输入”命令设备应该接收这些数据包和分析有效的载荷

确定一个绑定方案以至设备能够正确的交换数据包。

# 10.5、温度传感器实验

## 10.5.1、实验介绍

传感器节点采集温度和电池电压，并发送这些数据到中心收集节点进行处理。这里为了实验简单，只有一个中心节点收集这些信息，处理后通过串口送到计算机，可以在串口调试工具或超级终端上看到，为了提高网络的负载，可以增加中心收集节点。

这个实验必须做到：

- 自动形成一个网络
- 传感器设备必须能自动加入网络，并自动完成绑定
- 如果传感器设备没有从中心节点收到应答，它将自动移除到该中心节点的绑定。然后它将自动的去发现新的中心节点绑定。

该实验有两种设备被配置：传感器和中心收集设备（SimpleSensor 和 SimpleCollector）。

该实验只有一个命令--- SENSOR_REPORT_CMD_ID，

注意：温度和电池电压值不是很准确，许多硬件参数需要校准，看源代码传感器的相关校准参数。另外 2430 内部的温度传感器经实验检验不准确，可以采用外部温度传感器，不过这里为了实验方便，仍然采用内部温度传感器。

## 10.5.2、原理简要分析

加入网络之后，传感器设备将试图发现和绑定它自己到一个中心收集设备，如果发现多

个收集设备，它将挑选第一个响应的收集设备建立绑定；如果没有发现收集节点，他将周期性的继续搜索。

网络启动建立成功后，中心收集设备必须进入允许绑定模式，才能对传感器发送的绑定请求作出响应。在该实验中，通过按下收集设备的 S1，使之进入允许绑定模式，在该模式下，LED1 处于点亮状态；通过按下 S4，可以退出允许绑定模式，此时 LED1 关闭。

### 10.5.3、数据包发送和接收

绑定建立成功之后，传感器设备将根据定义的时间间隔周期的采集温度传感器和电池电压值，分别通过报告命令发送给收集设备。

收集节点接收到传感器设备发送的数据包后，它能显示到 PC 机或 LCD 上。该实验采用的是串口传输到 PC 机，通过串口调试工具可以观察到。

这时就可以在串口观察到结果：




从图中可以看出，该实验采用了一个中心节点，两个传感器设备，短地址分别为 0x796F，0x7970，可以看到两个设备环境中的温度和电池电压。注意：这里温度传感器采用 CC2430 内部温度传感器，不准确。

**备注：关于本实验更为详细的介绍请参阅教材：**

**《zigbee2006 无线网络与定位实战》**

## 10.6、灯开关实验操作流程图解

### 10.6.1、路径设定

首先把刚盘例子拷贝到路径：

C:\Texas Instruments\ZStack-1.4.2-1.1.0\..............................下；

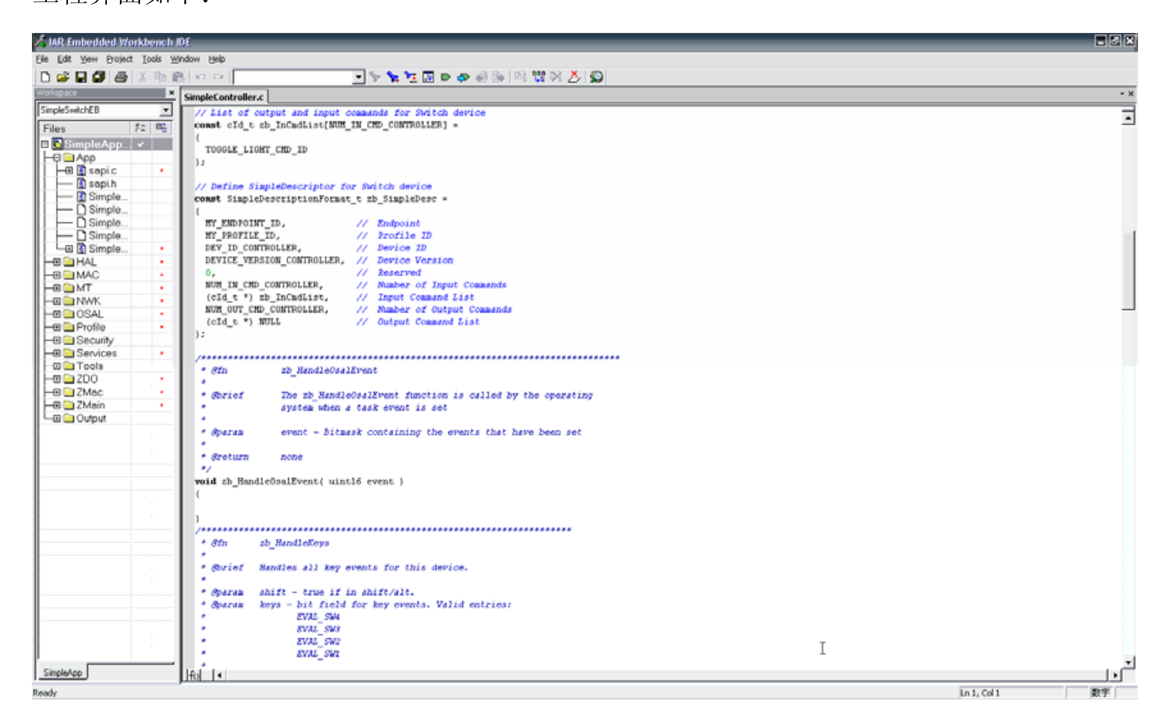
但有的客户需要放到 D:\Texas Instruments\ZStack-1.4.2-1.1.0\..............................下；主要取决于 IAR 安装路径。

## 10.6.2、打开工程：

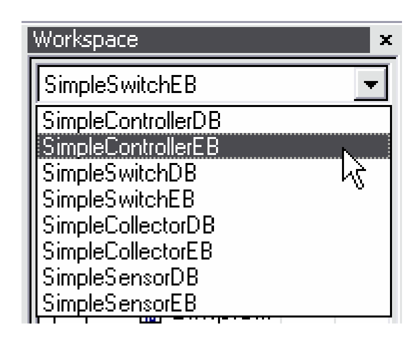
C:\Texas Instruments\ZStack-1.4.2-1.1.0\Projects\zstack\Samples\SimpleApp\CC2430DB



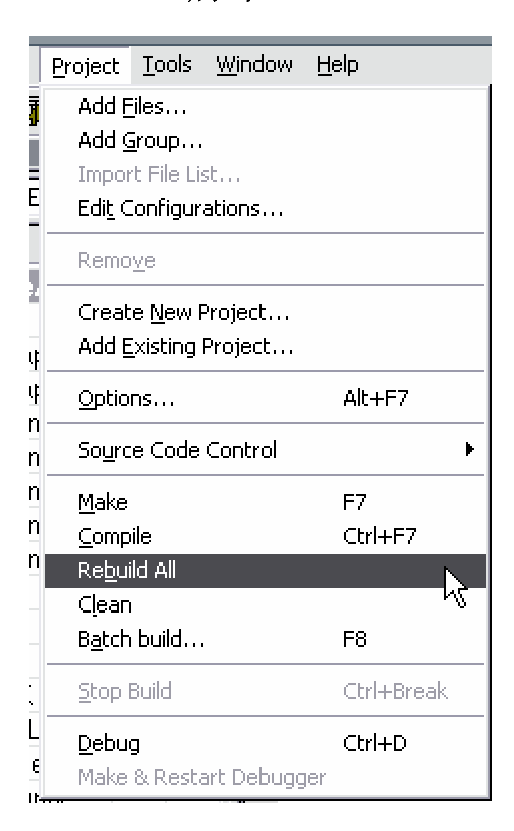

工程界面如下：



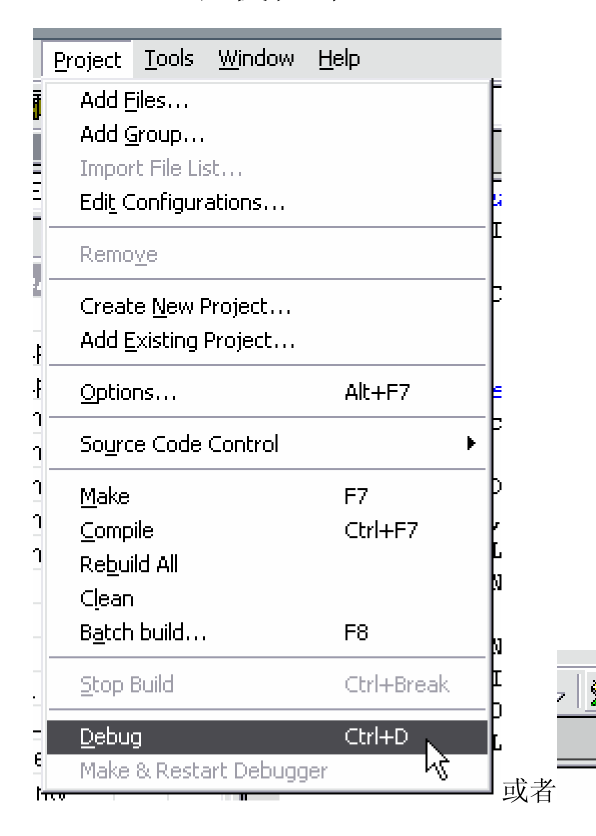
## 10.6.3、选择灯设备工程：



# 10.6.4、编译：

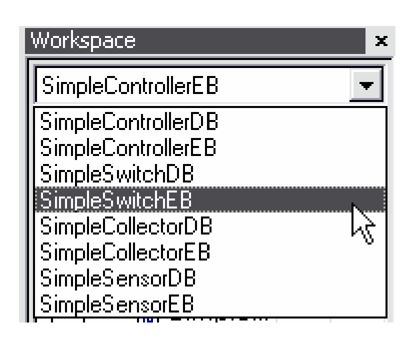



# 10.6.5、下载程序




或者

## 10.6.6、选择开关设备编译下载：




最后可以通过 4.2 节操作。

# 10.7、温度传感器实验操作流程图解

## 10.7.1、路径设定

首先把刚盘例子拷贝到路径：

C:\Texas Instruments\ZStack-1.4.2-1.1.0\................................下；

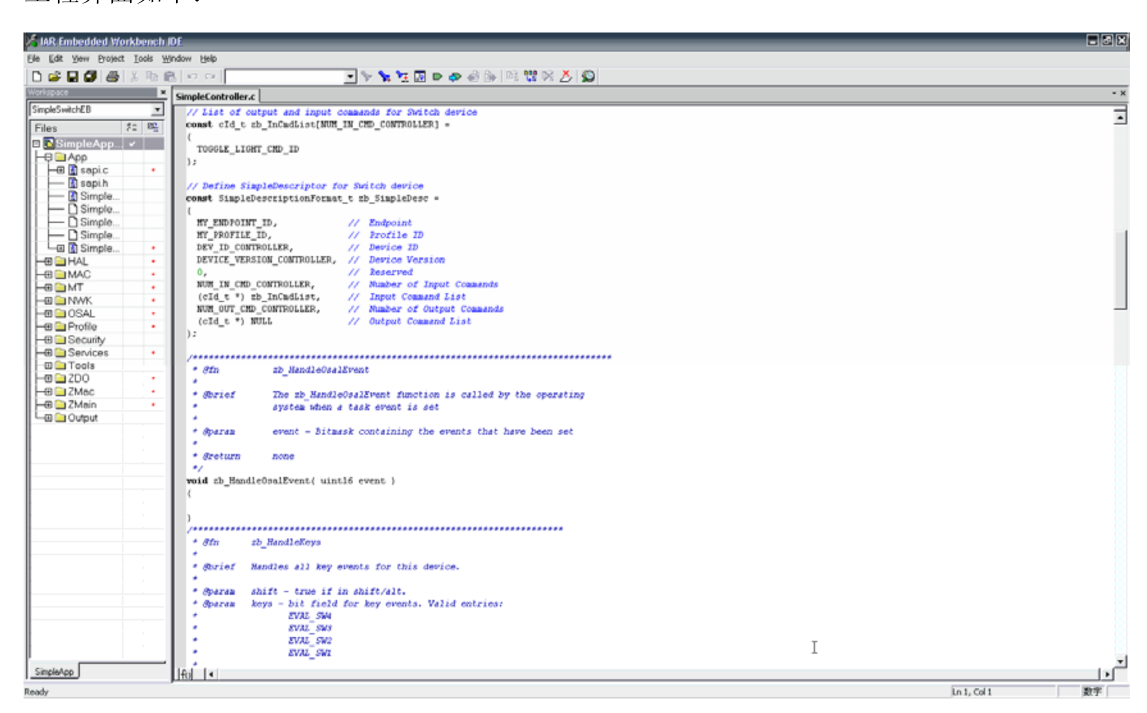
但有的客户需要放到 D:\Texas Instruments\ZStack-1.4.2-1.1.0\................................下；主要取决于 IAR 安装路径。

## 10.7.2、打开工程：

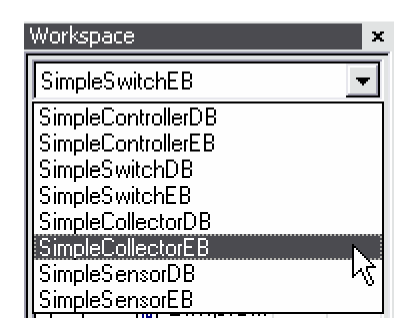
C:\Texas Instruments\ZStack-1.4.2-1.1.0\Projects\zstack\Samples\SimpleApp\CC2430DB



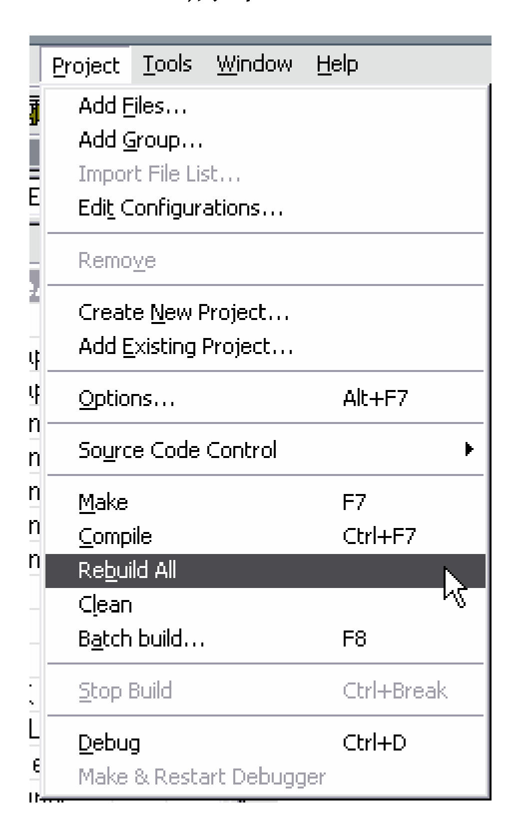

工程界面如下：



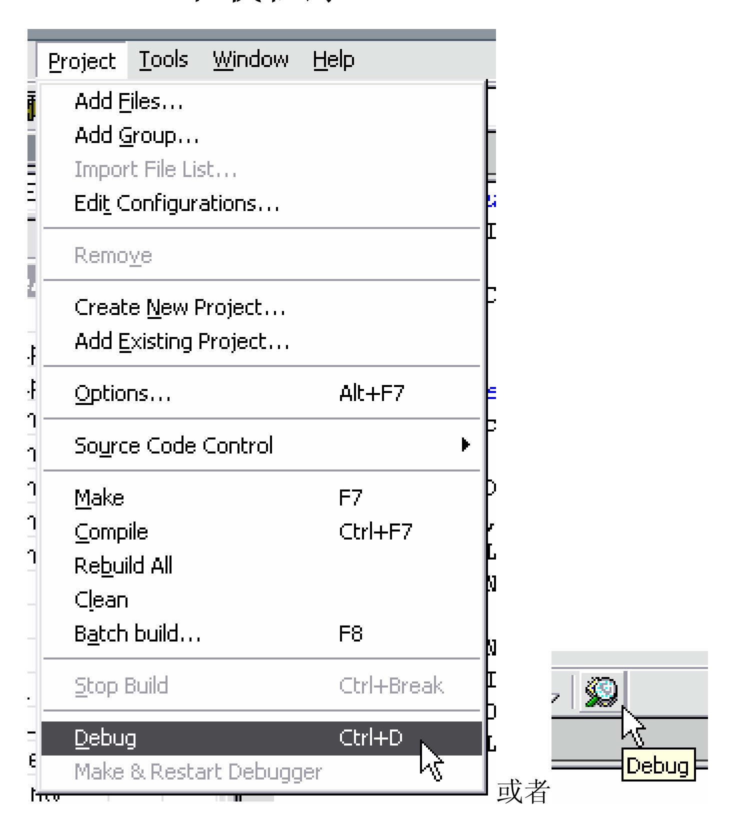
## 10.7.3、选择收集设备工程：




## 10.7.4、编译：

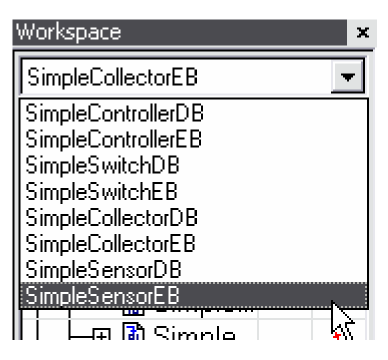


## 10.7.5、下载程序




或者

## 10.7.6、选择温度采集设备编译下载：




## 10.7.7、演示工作

将收集节点的串口与电脑 RS232 连接：

打开串口调试助手按照 5.3 节图片设置，就能看到收集温度传感器效果。

# 11、SimpliciTI 网络实验

## 11.1 实验目的

了解小型网络的实验方法，利用这种可靠性高、体积小的小型网络做无线数据传输，了解他们的用法，网络的结构等。

## 11.2 实验内容

利用 SimpliciTI 网络协议实现点对点对等网络的数据传输，再整个流程中共有，发现网络、连接网络、加入网络和数据传输过程，在 SimpliciTI 网络协议中已经将发现、连接、加入网络的功能全部包含，我们只需要编写我们需要的应用部分就可以了。

## 11.3 实验设备

- C51RF-3-PK/C51RF-3-JKS(两个 CC2430 模块、电源、USB 线)
- IAR 集成开发环境
- C51RF-3 仿真器

## 11.4 实验原理

德州仪器 (TI) 推出的针对简单小型 RF 网络的专有低功耗 RF 协议——SimpliciTI 网络协议。SimpliciTI 网络协议能够简化实施工作，尽可能降低微控制器的资源占用。在 CC1110/CC2510/CC2430 等片上系统 (SoC) 或 MSP430 超低功耗微控制器与 CC110x/CC2500 RF 收发器上运行。设计得当的 RF 协议对降低最终应用的功耗至关重要。最新 SimpliciTI 网络协议将支持客户开发超低功耗系统，同时降低系统成本，加速产品上市进程。

小型低功耗 RF 网络通常包含电池供电的设备，这就需要较长的电池使用寿命，以及较低的数据速率与占空比，而且直接相互通信的节点数量也非常有限。利用 SimpliciTI 网络协议可实现 MCU 资源占用的最小化，从而降低了低功耗 RF 网络的系统成本。需要路由功能的更复杂的网状网络通常需要 10 倍之多的程序存储器与 RAM。

尽管所需的资源不多，但 SimpliciTI 网络协议依然能够支持点对点通信，这种选择方案不仅可使用数据中心(Access Point)来存储并发送消息，还能通过范围扩展设备 (range extender) 来扩大网络覆盖范围以支持四次网络跳转。今后推出的版本还将添加频率捷变 (frequency agility) 等更多先进功能。

SimpliciTI 网络协议专为简单的 RF 网络而设计，对适合网状路由与标准化配置的大型网络的 ZigBee 而言是一种很好的补充。

SimpliciTI 网络协议以免版税、免许可费的源代码形式提供。欢迎开发人员根据各自具

体应用需求修改该协议。

SimpliciTI 网络协议支持各种低功耗应用，如报警与安全（烟雾探测器、玻璃破碎检测器、一氧化碳探测器、光传感器）、自动读表（气表、水表）、工业控制、家庭自动化（家电设备、车库开门器、环境设备）以及有源 RFID 等。

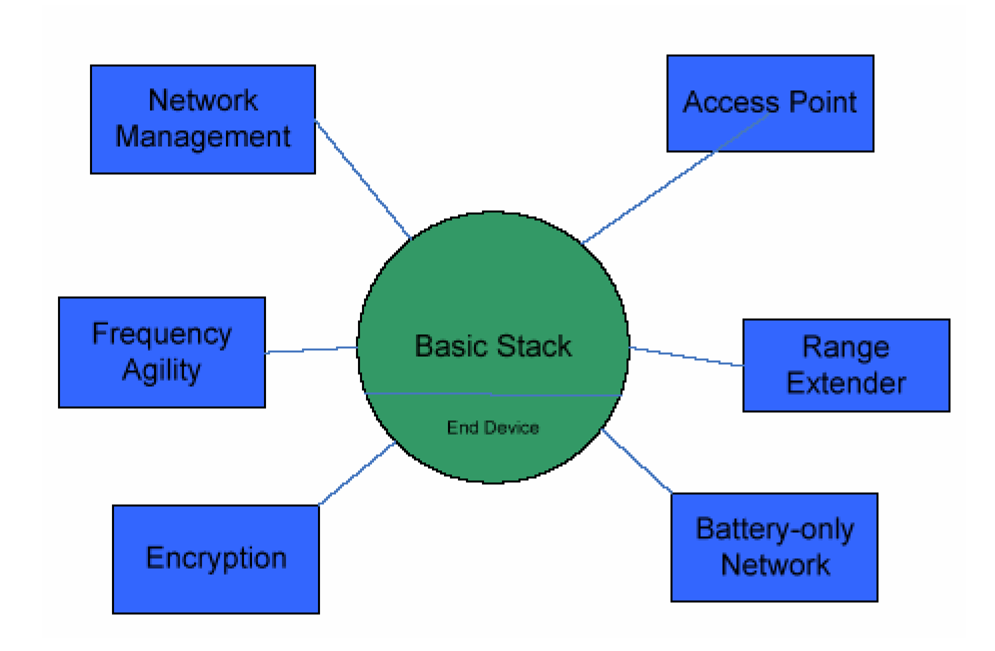



**图 1 SimpliciTI 协议的模块**

SimpliciTI 协议的模块如图 1 所示，包括 Network Management(网络管理)、Access Point(数据中心)、Frequency Agility(跳频)、Range Extender(范围扩展)、Encryption(加密)、Battery-only Network(低功耗网络)。

SimpliciTI 网络协议提供给应用层同级间的信息交流。这同级设备可以是传感器设备也可以是控制设备，也可以全部都是传感器接点，协议中没有作严格的区分。该协议的目的是让用户随心所欲的将自己想要的设备连接在一下进行信息传递。

从图 2 可以看到 SimpliciTI 网络协议主要包括如下三层：Application Layer(APP)应用层，Network Layer(NWK)网络层，Lite Hardware Abstraction Layer(LHAL)硬件逻辑层。SimpliciTI 网络的加密在网络层处理。

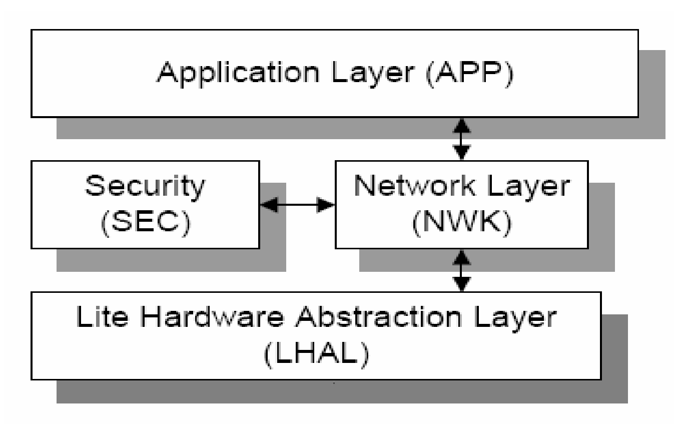



**图 2 SimpliciTI 网络协议结构**

从图 3 可以看到 SimpliciTI 的硬件逻辑层主要包括如下各层：射频层(Radio)、应用板支

持层程序包(BSP)。

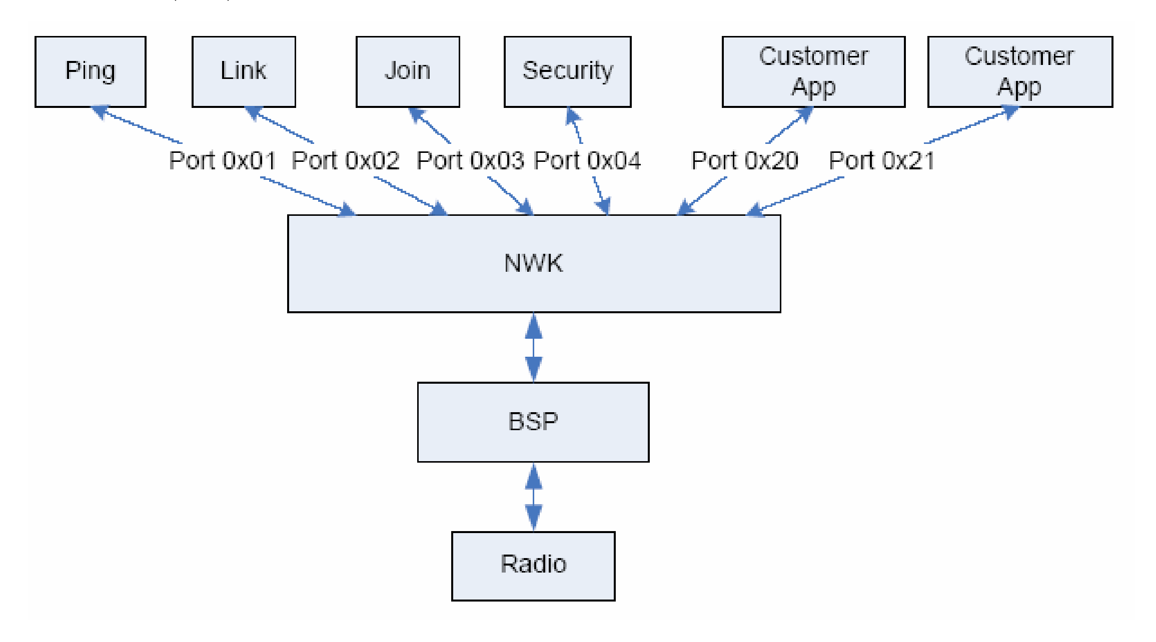



**图 3 SimpliciTI 网络协议的端口**

SimpliciTI 没有物理层（PHY）跟数据链路层（MAC/LLC），数据是直接从射频层接收过来，射频层的责任就是直拉将数据接收到。

协议内提供一个射频层(Radio)与网络层(NWK)连接的应用板支持层程序包(Board Support Package，BSP)，BSP 提供射频层与网络层通信的 SPI 接口。BSP 并不是传统意义上的硬件网络逻辑层，它仅仅是射频层与网络层之间通信的服务支持。

BSP 方便的提供了 LEDS 和 KEYS 以及 GPIO 引脚的定义，但是其它相关的定义并没有提供如：串口，LCD，定时器等，用户需要根据自己的程序自己定义。

网络层负责如下功能管理：

1)频段管理。

2)跳频支持。

3)调制方式、数据传输速率等无线参数管理。

4)加密管理。

5)数据传输。

6)CCA(清除信道评估)。

7)网络 ID。

8)设备地址。

9)加入、连接网络。

网络层管理射频发送与接收，并指定目的地址，目的地址通过端口号（PORT）指定，网络层是不会作任何帧处理的。

端口(PORT)概念根 TCP/IP 中的 PORT 概念相似，它是地址概念的延伸。端口编号范围是 0X 01 至 0X3F，如图 3 所示，0X 01 至 0X 1F 为端口，0X 20-0X 3F 为用户定义。端口用于网络层自身对网络的管理，这些端口不能被用户应用层应用。

就像 TCP/IP 中的 IP 一样必需对应相应的物理地址，网络层会在连接过程中把端口号跟

地址关联起来。

**NWK applications 网络应用层**

NWK applications 网络应用层提供网络层管理，除了提供外部 PING 访问以外还提供了很多供用户开发的接口 PORT，表 1 详细列出了相关的应用及描述。

**表 1 PORT 接口**

| Application | Port | Description |
|---|---|---|
| Ping | 0x01 | Just like the TCP/IP application. Echoes the received payload back to the sender. Direct addressing only. |
| Link | 0x02 | Used to associate two peers on different devices. |
| Join | 0x03 | Used when an Access Point is present to gain access to peers. |
| Security | 0x04 | Used to exchange security information such as keys. |
| Freq | 0x05 | Used to manage frequency migration to support frequency agility.. |
| Mgmt | 0x06 | General use **NWK** management application. Used for example as the poll port. |

对等层面结构的特点

SimpliciTI 网络协议对等层面的特点是只有网络层和应用层，如下表 2。应用层又分为两个部分网络应用层及用户程序应用层即 NWK applications 和 Peer applications。

表 2　SimpliciTI 网络协议对等层面的特点

| 层 | 连接种类 | 应答信息 |
|---|---|---|
| 网络层 | 连接 | 无 |
| 网络应用层 NWK applications | 连接 | 有 |
| 用户应用层 Peer applications | 双向信息交换 | 用户控制 |

网络结构

SimpliciTI 网络协议支持 2 种基本网络拓扑结构，其一为星状网络拓扑，它包括一个数据中心(Access Point)，数据中心主要负责网络管理。数据中心为终端节点(End Devices)提供数据存储、转发等，并管理网络内设备成员权限、连接权限以及安全等。数据中心还可以支持终端设备的功能扩展，如在网络中它可以自动实例化终端设备的传感器。

SimpliciTI 网络协议支持一个点对点网络。

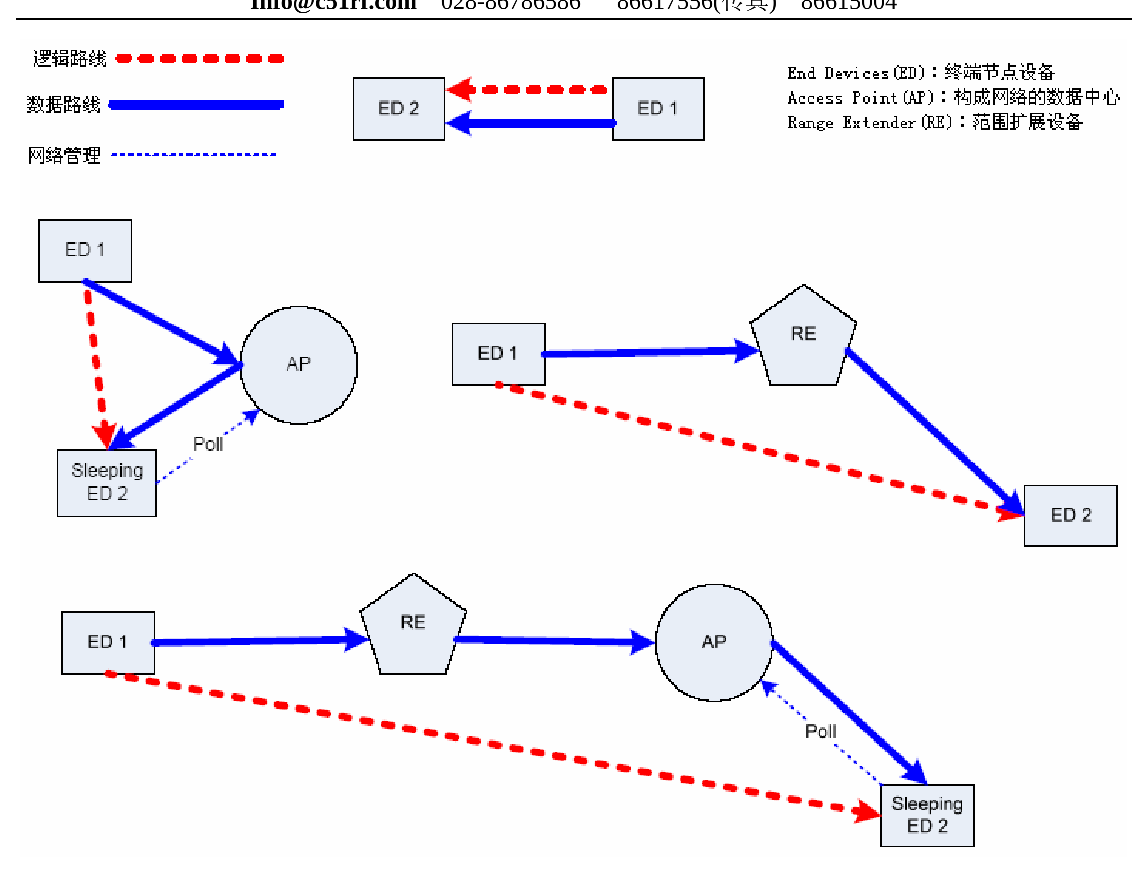



**图 4 点对点网络示意图**

SimpliciTI 协议应用是点对点无线数据传输。在许多情况下，许多节点共同连接在一起组成网络，但 SimpliciTI 可以支持 1 对点对点之间的数据传输，这对点对点设备不是必须设置也不是预先安排好的。如一个网络，任意 1 对设备都可以进行数据传输。

在 SimpliciTI 网络中，可以通过 AP 或 RE(有关 AP、RE 的介绍查阅本书的“设备”内的介绍)来扩展网络，如图 4 所示。

SimpliciTI 网络协议还支持一个网络拓扑扩展，即在星状网络的基础上，使用范围扩展设备扩展为串状网络拓扑。在此网络中，Range Extenders 范围扩展设备不对网络进行路由管理，它只作为一个数据收发中继站，范围扩展设备接受到数据后通过对数据包分析，如果发现目的地址不是其，则立即转发。

**设备**

SimpliciTI 网络提供三类设备分别是 End Devices 终端节点设备、 Access Points 数据中心、 Range Extenders 范围扩展设备。SimpliciTI 网络所说的设备是一个逻辑上的设备而并非硬件设备，即一个硬件设备可以拥有终端节点设备功能，也可以拥有数据中心功能。或拥有范围扩展设备功能，也可以拥有终端节点设备功能。

End Devices(ED)：终端节点设备，可以是传感器节点也可以是控制节点，该硬件模式可以用电池供电。

Access Point(AP)：构成网络的数据中心，同一个网络中 Access Points 可以和终端设备共存，它可以组成一个网络，挂接多个传感器设备或者控制设备，同一个网络中也允许有两个 Access Point。在特殊模式下它可以接收所有能够接收到的数据，包括通过范围扩展得到的数据。

Range Extender(RE)：范围扩展设备，有目的地对网络覆盖范围进行扩展，这是一个常开设备。它的主要功能是重复发送从发送设备过来的数有目的的让发送设备的影响范围扩展。在一个网络中要使网络稳定请将范围扩展限制在 4 个以内。范围扩展运行在混杂模式可以收到它能收到的所有数据。

**地址**

一个网络地址由两个部分构成：一个物理地址（由程序设置）和一个应用层地址即PORT。

物理地址是在程序编译的时候就已经设定;网络中每一个设备必需要分配网络中唯一的硬件物理地址。物理地址的长度限制在四个无符号字节以内。而且 CC2430 第一个地址字节不能是 0x00 或者 0xff。在这两个射频芯片工作时帧格式已经定了第一个字节 0X00 和 0xff 会被认为是广播帧。

应用程序接口地址（PORT）是在设备加入网络，网络连接的过程序中分配，是不受用户干涉的。

**网络连接**

协议确定两个设备之间通信会有一个连接过程。连接信息包含一个连接标志 4 个字节，接收连接的设备才能允许该设备加入网络。当然允许设备加入网络的数量受到用户随机读写存储器 RAM 区与 PORT 地址限制。

**数据帧格式**

| 报头 | 同步 | 长度 | 目的地址 | 源地址 | PORT | 设备信息 | 交换记录 | 有效数据 | 效验 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 4 | 1 | 4 | 4 | 1 | 1 | 1 | n | 2 |

CRC

数据帧大小为至小 22 个字节，最大为 74 个字节。SimpliciTI 对其中数据长度、目的地址、源地址、PORT(端口中)、设备信息、交换记录、有效数据等进行 CRC 校验。有效数据载荷为大于等于 0 小于等于 52 个字节。

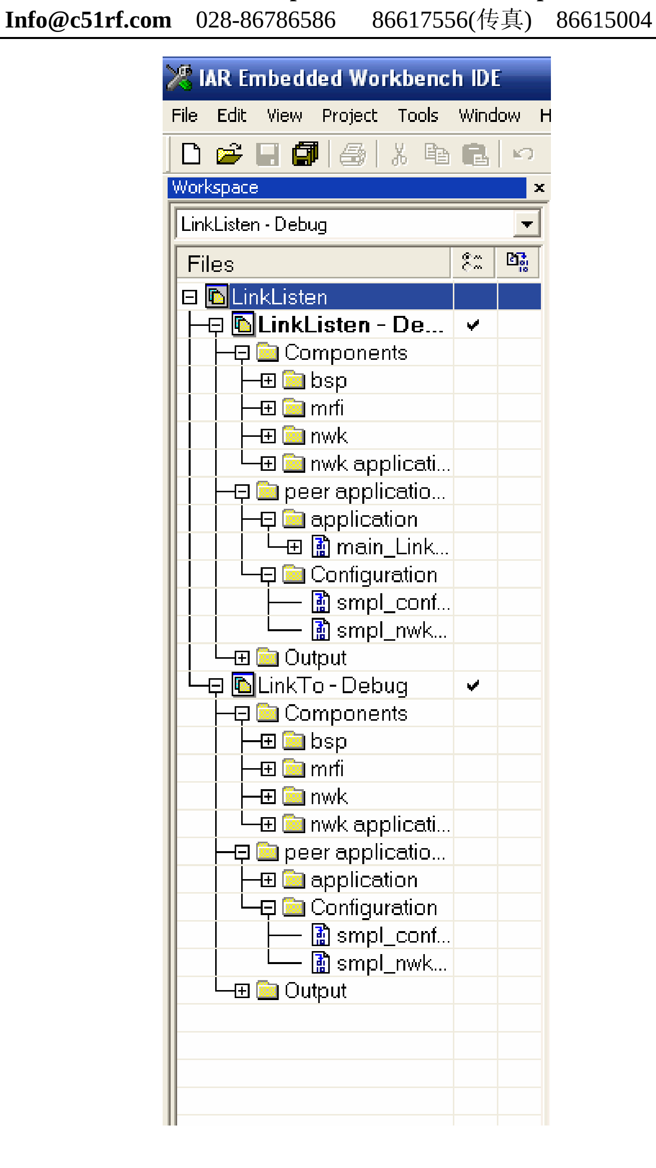
# 11.5 数据传输的实现

本实验工程共包括 3 个文件夹，分别是：

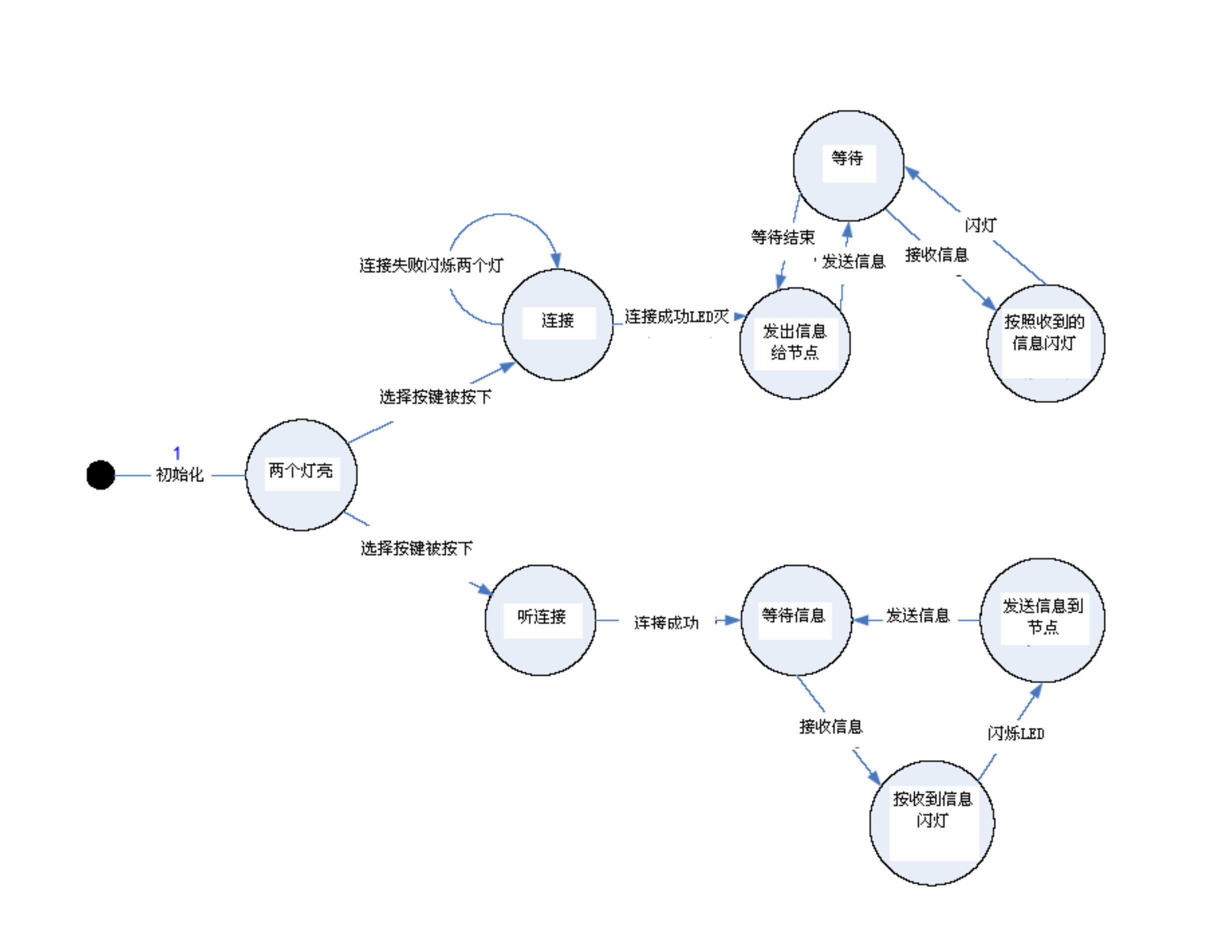
文件夹 Components：基于系统的 SimpliciTI 协义栈工程组成文件；

文件夹 Documents：基于系统的 SimpliciTI 协义栈工程说明文档；

文件夹 Projects：基于系统的 SimpliciTI 协义栈的点对点通信工程文件及配置文件。在此路径下 Projects\Examples\Peer applications\CC2430DB\2 simple end devices with bi-di 工程文件“LinkListen.eww”安装 IAR7.20H 或以上版本用户双击该文件即可打开工程。此文件夹为本章实验所用的工程文件、源代码。打开工程后，可以看到整个工程的结构和工程中存在的源代码。


IAR Embedded Workbench IDE
File Edit View Project Tools Window H
Workspace
LinkListen - Debug
Files
LinkListen
LinkListen - De...
Components
bsp
mrfi
nwk
nwk applicati...
peer applicatio...
application
main_Link...
Configuration
smpl_conf...
smpl_nwk...
Output
LinkTo - Debug
Components
bsp
mrfi
nwk
nwk applicati...
peer applicatio...
application
Configuration
smpl_conf...
smpl_nwk...
Output

## 11.6 流程图




## 11.7 程序的实现

网络状态枚举

用户可以跟据返回状态判断网络的工作情况，如程序清单 6.1 所示。

程序清单 6.1 如下：

```
enum smplStatus  {
  SMPL_SUCCESS,//
  SMPL_TIMEOUT,//
  SMPL_BAD_PARAM,
  SMPL_NOMEM,
  SMPL_NO_CONNINFO,
  SMPL_NO_FRAME,
  SMPL_NO_LINK,
  SMPL_NO_JOIN,
  SMPL_NO_SLEEP,
  SMPL_NO_CHANNEL
};
```

网络连接函数

连接函数，不断等待。硬件(C51RF-PS 高频模块，如图 1.3 所示)表现为连接没有成功两个 LED 灯不断闪烁，直到连接成功，连接成功后关掉第一个灯，如程序清单 6.2 所示。

程序清单 6.2 如下：

```
static void linkTo()
{
  uint8_t   msg[2], delay = 0;
  while (SMPL_SUCCESS != SMPL_Link(&sLinkID1))//等待连接成功
  {
    // 闪烁 LED 灯，直到连接成功
    toggleLED(1);//闪烁 LED1
    toggleLED(2);//闪烁 LED2
    SPIN_ABOUT_A_SECOND;//随机延时
  }
  //连接成功关闭红色 LED，效换信息后闪烁黄色 LED
  if (BSP_LED1_IS_ON())//判断 LED1 当前状态是否为开
  {
    toggleLED(1);//取反 LED1 状态
  }
  // 填装 闪烁哪一个 LED 参数，这里先取 LED1 红色 LED
  msg[0] = 1;
  while (1)
  {
    SPIN_ABOUT_A_SECOND;//随机延时
    if (delay > 0x00)
    {
      SPIN_ABOUT_A_SECOND;
    }
    if (delay > 0x01)
    {
      SPIN_ABOUT_A_SECOND;
    }
    if (delay > 0x02)
    {
      SPIN_ABOUT_A_SECOND;
    }
    // delay longer and longer -- then start over
    delay = (delay+1) & 0x03;
    // 填装信息交换次数
```

```
        msg[1] = ++sTxTid;
        SMPL_Send(sLinkID1, msg, sizeof(msg));//发送信息
    }
}
```

等待连接函数

本章节程序只等待一个有效连接，谁首先进入连接谁最先得到响应。硬件表现：等待连接端连接 LED1 点亮。如果没有连接就常亮，连接成功两个灯熄灭。点对点成功交换数据表现为：两个设备各有一个 LED 灯同频闪烁，周期约为 4 秒，如程序清单 6.3 所示。

程序清单 6.3 如下：

```
static void linkFrom()
{
    uint8_t       msg[2], tid = 0;
    toggleLED(1);
     //等待连接
     SMPL_LinkListen(&sLinkID2);
     // 关掉 LED2
     toggleLED(2);
     //每收到一次信息 LED2 （黄灯）状态被取反一次
     *msg = 0x01;
     while (1)
     {
       //等待中断返回状态参数 接收信息
        if (sSemaphore)
        {
            *(msg+1) = ++tid;
            SMPL_Send(sLinkID2, msg, 2);//信息接收成功后返回信息
            sSemaphore = 0;
        }
     }
}
```

网络层连接函数根据不同的宏定义，会编译不一样的语句，得到不一样的程序执行情况；本章讲述的点对点通信；涉及到的是终端节点设备，定义 END_DEVICE，如程序清单 6.4 所示。

程序清单 6.4 如下：

```
smplStatus_t SMPL_LinkListen(linkID_t *linkID)
{
#if defined( END_DEVICE ) && defined( USE_ADDRESS_FILTERING )
    //   不用地址匹配则用广播的形式发送信息。
    // 这只适合对终端节点设备通信如果需要地址匹配接收程序才会判断地址
```

```
   MRFI_DisableRxAddrilter();//关地址匹配 以更实现加入网络，实现连接过程
#endif    // END_DEVICE
  nwk_setListenContext(LINK_LISTEN_ON);
  while (!(*linkID=nwk_getLocalLinkID()));
  if (!(*linkID))
  {
    nwk_setListenContext(LINK_LISTEN_OFF);
  }
#if defined( END_DEVICE ) && defined( USE_ADDRESS_FILTERING )
 //   不用地址匹配则用广播的形式发送信息。
  // 这只适合对终端节点设备通信如果需要地址匹配接收程序才会判断地址
  MRFI_EnableRxAddrFilter();//开地址匹配
#endif
  return SMPL_SUCCESS;
}
网络连接函数
smplStatus_t nwk_link(linkID_t *lid)
{
  uint8_t           msg[sizeof(sLinkToken)+4];
  connInfo_t    *pCInfo = nwk_getNextConnection();
  if (pCInfo)
  {
    addr_t          addr;
    union
    {
      ioctlRawSend_t        send;
      ioctlRawReceive_t recv;
    } ioctl_info;
    if (!nwk_allocateLocalRxPort(LINK_SEND, pCInfo))
     {
       nwk_freeConnection(pCInfo);
      return SMPL_NOMEM;
    }
    memset(addr.addr, 0x0, NET_ADDR_SIZE);
    ioctl_info.send.addr = &addr;
    ioctl_info.send.keep = FI_INUSE_UNTIL_DEL;
    ioctl_info.send.msg    = msg;
    ioctl_info.send.len    = sizeof(msg);
    ioctl_info.send.port = SMPL_PORT_LINK;
```

```
    // 复制 link token 到需要发送的数据中
    memcpy(msg+1, &sLinkToken, sizeof(sLinkToken));
    //设置已经由网络分配的端口号
    msg[sizeof(sLinkToken)+1] = pCInfo->portRx;
    // 设置连接数量
    msg[sizeof(sLinkToken)+2] = sLinkNumber;
    // 设置接收方式
    msg[sizeof(sLinkToken)+3] = nwk_getMyRxType();
    // 设置 APP 信息类型
    *msg = sizeof(msg) - 1;
    SMPL_Ioctl(IOCTL_OBJ_RAW_IO, IOCTL_ACT_WRITE, &ioctl_info.send);
#ifndef RX_NEVER
    {
      uint8_t spin = NWK_RX_RETRY_COUNT;
      ioctl_info.recv.port = SMPL_PORT_LINK;
      ioctl_info.recv.msg    = msg;
      ioctl_info.recv.addr = (addr_t *)pCInfo->peerAddr;
      do
      {
        if (SMPL_SUCCESS == SMPL_Ioctl(IOCTL_OBJ_RAW_IO, IOCTL_ACT_READ, &ioctl_info.recv))
        {
          break;
        }
        if (!spin)
        {
          // invalidate connection object
          nwk_freeConnection(pCInfo);
          nwk_txDone(FI_INUSE_UNTIL_DEL, SMPL_PORT_LINK);
          return SMPL_NO_LINK;
        }
        NWK_DELAY(WAIT_LONG_TIME);
        --spin;
      } while (1);
      //存放连接数
      pCInfo->linkNum = sLinkNumber;
      // 连接应答端口
      pCInfo->portTx    = *(msg+1);
      *lid              = pCInfo->thisLinkID;        // return our local port number
      if (*(msg+2) == F_RX_TYPE_POLLS)
```

```
        {
          pCInfo->hops2target = MAX_HOPS;
        }
        else
        {
          // can't really use this trick because the device could move...
          // pCInfo->hops2target = MAX_HOPS - ioctl_info.recv.hopCount;
          pCInfo->hops2target = MAX_HOPS;
        }
      }
#endif   // !RX_NEVER
      ++sLinkNumber;
      if (!sLinkNumber)
      {
        sLinkNumber = 1;
      }
      return SMPL_SUCCESS;
    }
    return SMPL_NOMEM;
}
```

网络层函数

下面介绍网络层内本章所涉及到的关键函数。

nwk_join.c：程序块。

sLinkToken = 0xDEADBEEF：连接标志。

static void smpl_send_join_reply(mrfiPacket_t *frame)：加入应答。

void saveAddress(mrfiPacket_t *frame)：AP 存放加入进来的终端设备地址。

uint8_t isJoined(mrfiPacket_t *frame)：判断设备是不是重复加入网络。

smplStatus_t nwk_join(void)：网络加入函数。

nwk_globals.c：程序块。

void nwk_globalsInit(void) ：初始化该设备网络中的四字节地址，将存放于 ROM 中的地址调入 RAM 中方便以后调用。

addr_t const *nwk_getMyAddress(void)：取出本机地址 返回指向地址的指针。

void nwk_setMyAddress(addr_t *addr)：用户程序手动设置该设备在网络中的地址。

void nwk_setAPAddress(addr_t *addr)：设置加入到的网络的地加入过程中读回的 AP 的地址。

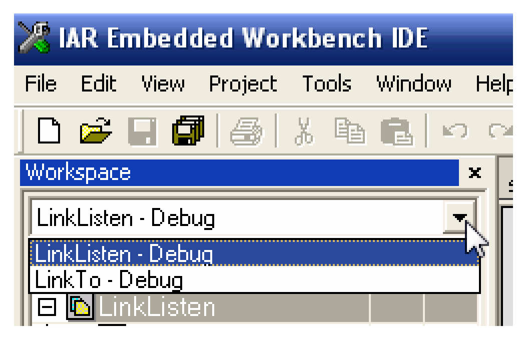
addr_t const *nwk_getAPAddress(void)：取回设备网络的地址，即设备所在 AP 的地址。

MRFI_SetRxAddrFilter((uint8_t *)nwk_getMyAddress())：设置接收匹配地址。

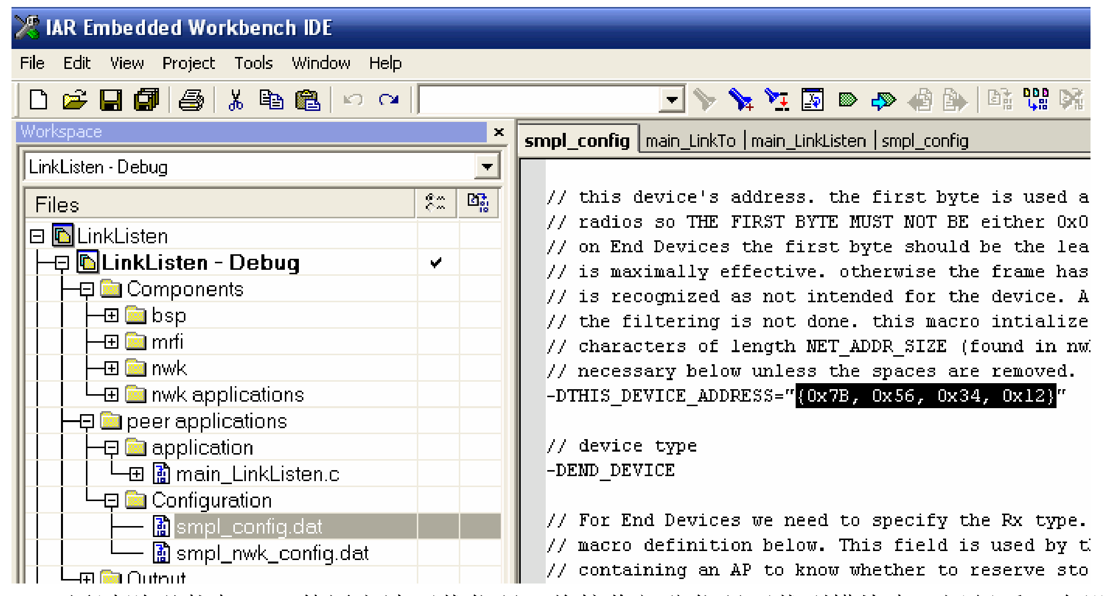
MRFI_EnableRxAddrFilter()：打开接收地址匹配。

# 11.8 实验步骤

11. 连接好硬件，请参阅系统说明书。
12. 打开 IAR 工程后，可以看到如下图所示的窗口，在本窗口中可以看到，这个工程中共分为两个模块，LinkListen 和 LinkTo。首先选择 LinkListen。




13. 设定设备的地址，设备地址在一个网络中是唯一的，它的设定在 smpl_config.dat 文件中，选择相应的 smpl_config.dat 文件。文件中的-DTHIS_DEVICE_ADDRESS 就是设备的地址，我们需要设置的是第一项，可以设置为 0x00~0xFE，共 255 个设备。对于网络中的每一个设备都必须设置自己的唯一的设备地址。




14. 通过说明书中 IAR 使用方法下载代码，将接收部分代码下载到模块中。运行后，在没有节点加入进来之前，红灯点亮。
15. 用同样的方法下载 LinkTo，程序运行后，在没有加入网络前，小灯同时闪烁，到加入网络后，黄灯开始闪烁，相应的红灯也开始闪烁，这样两个设备就开始通讯了。值得注意的是，这个网络没有自恢复功能，每一次运行都必须要重启 LinkTo。